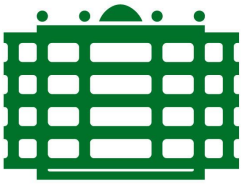

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Fakultät für Informatik
Professur Informationssysteme und Softwaretechnik

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Informatiker
eingereicht von

Jöran Toschev

Entwurf und Implementierung einer neuen Architektur für TESSI

Prüfer: Prof. Dr.-Ing. Petr Kroha
Betreuer: Lars Rosenhainer M. A.

Chemnitz, d. 27. August 2003

Technische Universität Chemnitz
Fakultät für Informatik

Aufgabenstellung für die Diplomarbeit

Name, Vorname: Toschev, Jöran
Matrikelnummer: 06820

Thema

Entwurf und Implementierung einer neuen Architektur für TESSI

Zielstellung

Die Arbeit hat zum Ziel, eine neue Version des Programmes *Textual Assistant* (TESSI) zu realisieren. Ausgangspunkt ist dabei die aktuelle Version 1.1 von TESSI, deren Funktionsumfang erhalten bleiben soll. Die neue Version soll sich durch eine klare Strukturierung und vollständige Dokumentation auszeichnen.

Zur Realisierung sollen Methoden und Werkzeuge des Reengineering auf ihre Verwendbarkeit in der Arbeit untersucht und angewendet werden.

Schwerpunkte der Diplomarbeit:

1. Einführung in Methoden und Werkzeuge des Reengineering
2. Analyse des Ist-Standes im Projekt TESSI, Verbesserungsmöglichkeiten und Anwendbarkeit von Reengineering-Werkzeugen
3. Entwurf einer neuen Architektur
4. Lauffähiges Programm gemäß der entworfenen Architektur
5. Dokumentation für Benutzung und Erweiterung der neuen Version
6. Einschätzung der Arbeit und Ausblick auf die Weiterentwicklung

Betreuender Hochschullehrer: Prof. Dr.-Ing. Petr Kroha
Fakultät: Informatik
Professur: Informationssysteme und Softwaretechnik

Beginn am: 1. Mai 2003
Einzureichen am: 31. Oktober 2003

Selbständigkeitserklärung

Hiermit erkäre ich, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Die Diplomarbeit wurde noch keiner Prüfungsbehörde in dieser oder anderer Form vorgelegt.

Chemnitz, d. 27. August 2003

Jöran Toschev

Danksagung

An dieser Stelle möchte ich mich bei allen Menschen, die bei der Entstehung meiner Diplomarbeit mitwirkten oder diese durch ihre Hilfe und Unterstützung erst möglich gemacht haben, von ganzem Herzen bedanken.

In erster Linie gilt mein Dank Professor Petr Kroha und Lars Rosenhainer, die mich in Studium und Diplomarbeit unterstützten und die mir stets mit Rat und Tat zur Seite standen.

Beistand, Anteilnahme und Unterstützung habe ich von meiner Frau Marijke und meinen Eltern Anita und Jordan Toschev erfahren. Sie waren da für mich an jedem einzelnen Tag und auch dann, wenn es nur schwer vorwärts ging. Viel Freude und Kraft schenkte mir ein neues Licht in meinem Leben, meine Tochter Astrid.

Danken möchte ich meinen Kommilitonen und Freunden, von denen ich manchen Rat und manche Hilfe erhielt.

Erwähnen möchte ich Frau Martina Wegert von Rational Software (IBM Deutschland GmbH) sowie die Mitarbeiter im Netbeans MDR Projekt für ihre Arbeit und unkomplizierte Hilfe. Ich möchte auch die vielen Mitwirkenden an Open-Source-Projekten nicht vergessen, deren Software ich nutze und ohne deren Arbeit meine Arbeit nicht denkbar gewesen wäre.

INHALTSVERZEICHNIS

1. Einleitung 8

2. Einführung in das Reengineering 10
2.1 Einordnung und Begriffsbestimmung 10
2.2 Einige Prozesse im *Reengineering* 13
2.2.1 Programmverstehen 13
2.2.2 Messen 15
2.2.3 Restrukturierung — *Refactoring* 18
2.2.4 Wiederverwendung (*Reuse*) von Programmcode 19
2.3 Vorhandene Werkzeuge 22
2.3.1 Programmverstehen und -modifizieren 22
2.3.2 Metriken und Visualisierung 23
2.3.3 *Refactoring* 27
2.3.4 Sonstige 28

3. Softwarearchitektur 30
3.1 Begriffsbestimmung und Einordnung 30
3.2 Softwarearchitektur und *Refactoring* 32
3.3 Softwarearchitektur im Projekt TESSI 33

4. Vorbetrachtungen 35
4.1 Was bisher geschah 35
4.2 Geplante Veränderungen 36
4.2.1 Metamodelle, Modellierungsumfang, Datenaustausch . . . 36
4.2.2 Nutzerschnittstelle 37
4.2.3 Wartung 37
4.2.4 Dokumentation 37
4.3 Untersuchung der Architektur im aktuellen TESSI 38
4.3.1 *Reverse Engineering* 38
4.3.2 Modulaufbau 38
4.3.3 *Separation of Concerns* 42
4.4 Zielsetzung 49
4.4.1 Architektur 49
4.4.2 Implementierung 50
4.4.3 Dokumentation 50
4.4.4 Auswertung 50

5. Beschreibung der neuen Architektur . . . 51
5.1 Allgemeine Struktur und allgemeine Dienste . . . 51
5.1.1 Verfeinerte Modulstruktur . . . 52
5.1.2 Modulhandhabung . . . 53
5.1.3 Nachrichtensystem . . . 55
5.1.4 Graphische Komponenten . . . 58
5.1.5 Modulsteuerung aus der gemeinsamen Oberfläche . . . 60
5.1.6 Datenhaltung . . . 60
5.2 Modul Basis . . . 63
5.3 Modul Datenhaltung . . . 65
5.4 Modul Spezifikationstext . . . 68
5.5 Modul Modellübersicht . . . 70
5.6 Modul Modellmanipulation . . . 72
5.7 Modul Thesaurus . . . 73
5.8 Modul Textgenerierung . . . 74
5.9 Modul Hilfe . . . 75
5.10 Adapter-Bibliothek . . . 75

6. Standardimplementierung TESSI 2.0 . . . 76
6.1 Allgemeine Festlegungen . . . 76
6.1.1 *Build*-Umgebung . . . 76
6.1.2 Verzeichnisstruktur . . . 77
6.2 Modulhandhabung . . . 78
6.3 Nachrichtensystem . . . 79
6.4 Graphische Komponenten und Modulmenüs . . . 80
6.5 Modul Basis . . . 80
6.6 Modul Datenhaltung . . . 80
6.7 Modul Spezifikationstext . . . 81
6.8 Modul Modellübersicht . . . 82
6.9 Modul Modellmanipulation . . . 82
6.10 Modul Thesaurus . . . 82
6.11 Modul Textgenerierung . . . 83
6.12 Modul Hilfe . . . 83
6.13 Adapterbibliothek . . . 84

7. Einschätzung und Ausblick . . . 86
7.1 Analyse von TESSI 2.0 . . . 86
7.1.1 Modularisierung . . . 86
7.1.2 *Separation of Concerns* . . . 87
7.2 Zukünftige Verbesserungen . . . 90
7.2.1 Unterstützung verschiedener UML-Versionen . . . 90
7.2.2 Unterstützung verschiedener Thesauri . . . 91
7.2.3 Strukturierter Spezifikationstext . . . 91
7.2.4 Unterstützung von Metriken . . . 92
7.3 Fazit . . . 92

Abkürzungsverzeichnis . . . 94

Literaturverzeichnis . . . 96

Anhang 98

A. Standardereignisse 99

B. Benutzerhandbuch 108

C. Entwicklerhandbuch 109
C.1 Erstellung eines neuen Moduls 109
C.2 Überarbeitung bestehender Module 112

D. Quelltext 113

E. Programmübergabe 114

1. EINLEITUNG

Diese Diplomarbeit hat zum Ziel, eine neue Version des Programms *Textual Assistant* (TESSI) zu realisieren. TESSI entsteht im Rahmen eines gleichnamigen Forschungsprojekts der Professur Informationssysteme und Softwaretechnik (ISST) der TU Chemnitz.

TESSI dient als CASE[1]-Werkzeug speziell für die Phase der Erfassung und Analyse der Anforderungen in einem Softwareprojekt. Der Nutzer von TESSI kann einen Spezifikationstext textuell analysieren und entwickelt ein objektorientiertes Modell daraus. Der Spezifikationstext ist in der Sprache des Auftraggebers formuliert und gibt dessen Vorstellungen wieder. Das entwickelte Modell dagegen gibt die Vorstellung des Analytikers wieder. Aus diesem Modell lassen sich lesbare Texte in verschiedenen Formaten generieren, die dem Auftraggeber vorgelegt werden können. Findet dieser Widersprüche zu seiner Vorstellung oder eine unvollständige Repräsentation dieser Vorstellungen, können diese Fehler zeitig korrigiert werden. Als Ergebnis eines iterativen Prozesses entsteht ein Spezifikationstext und ein daraus abgeleitetes Modell der zu entwickelnden Software. Dieses Modell läßt sich nach Rational Rose®exportieren und kann dort weiterbearbeitet werden.

Die aktuelle Version 1.1 verfügt über einen Funktionsumfang, der den Einsatz in konkreten Industrieprojekten möglich macht. Pläne für Erweiterungen und einige Defizite führten aber zum Entschluß, den gegenwärtigen Stand zu überarbeiten. Diese Überarbeitung sollte vor allem zu folgenden Veränderungen führen:

- Einführung einer modularen Architektur, die die getrennte Bearbeitung und Weiterentwicklung von Funktionsbereichen ermöglicht (z. B. in studentischen Arbeiten, die nur ein bestimmtes Modul umfassen)
- Bereitstellung einer vollständigen Dokumentation für sowohl die Benutzung der Software (Benutzerhandbuch), als auch die Erweiterung des Programms (API[2]-Dokumentation)

Die neue Version von TESSI (im Folgenden TESSI 2.0 genannt) sollte selbst keine neue Funktionalität einführen, weshalb sich die vorliegende Arbeit in weiten Teilen auch als *Reengineering*-Aufgabe, sogar als *Reverse Engineering* auffassen läßt. Die Nutzung entsprechender Werkzeuge war daher ebenfalls ein vorhersehbar wichtiger Punkt.

Ein wichtiges Ergebnis der Arbeit sollte ein lauffähiges Programm sein. Dieses Programm sollte einerseits die gleichen Aufgaben erfüllen können, wie die ältere TESSI-Version. Darüber hinaus sollte und soll es die Bemühungen unterstützen, Erweiterungen leicht zu integrieren.

[1] Computer Aided Software Engineering
[2] Application Programming Interface

Parallel zu Entwurf und Implementierung lief ein anderes Projekt, das sich bereits auf das entstehende Programm stützte. Lars Rosenhainer brachte im Rahmen seiner innerhalb des TESSI-Projektes angestrebten Promotion viele Wünsche, Anregungen und Hinweise ein und leistete auch als erster Anwender wertvolle Dienste.

Die Diplomarbeit hat den folgenden Aufbau: Zunächst wird in Kapitel 2 ein Blick auf das *Reengineering* allgemein sowie auf einige Methoden und Werkzeuge geworfen. Im darauf folgenden Kapitel 4 wird der Stand im Projekt TESSI untersucht. Dabei wird auch auf die Anwendbarkeit von *Reengineering*-Werkzeugen eingegangen und es werden Verbesserungsmöglichkeiten und eine Zielstellung abgeleitet. Kapitel 5 beschreibt die Architektur von TESSI 2.0 und geht auf die einzelnen Module ein. Details zur Implementierung beschreibt Kapitel 6. Die Arbeit schließt mit einer Einschätzung und dem Ausblick auf kommende Veränderungen (Kapitel 7). Im Anhang befinden sich das Benutzerhandbuch (siehe Anhang B) und das Entwicklerhandbuch (siehe Anhang C).

2. EINFÜHRUNG IN DAS REENGINEERING

Eines der Ziele dieser Arbeit ist es, für eine bestehende Software eine verbesserte Architektur zu entwerfen. Tätigkeiten dieser Art werden üblicherweise mit dem Begriff *Reengineering* beschrieben. Dieses Kapitel soll in das Gebiet des *Reengineering* einführen.

Das Kapitel ist wie folgt aufgebaut: im nächsten Abschnitt (2.1) wird das *Reengineering* zunächst in den Kontext der Softwareentwicklung eingeordnet und es wird versucht, einige der Begriffe zu bestimmen. Abschnitt 2.2 untersucht Teilbereiche und Methoden. Der abschließende Abschnitt gibt einen kurzen Überblick einiger Werkzeuge, die verschiedene Tätigkeiten im Umfeld des *Reengineering* unterstützen.

2.1 Einordnung und Begriffsbestimmung

Im Lebenszyklus von Softwaresystemen nimmt die Wartung einen bedeutenden Platz ein, besonders durch ihren hohen Ressourcenverbrauch. So ist in [Kro97, S. 187] nachzulesen, daß für die Wartung mit einem Anteil von 60 % an den Gesamtkosten über die Lebenszeit eines Softwaresystems zu rechnen ist. Balzert nennt als eine Faustregel: „Der Aufwand für die Wartung & Pflege ist typischerweise um einen Faktor von 2 bis 4 größer als der Entwicklungsaufwand für ein umfangreiches Produkt.“ ([Bal00, S. 1093]). In [Mue97] werden Zahlen aus verschiedenen Studien genannt (40–80 % Anteil am Gesamtaufwand). Mit zunehmendem Alter von Software steigt in der Regel auch der Wartungsaufwand. Man könnte sagen, daß Wartung auch die maximale Lebensdauer eines Systems mitbestimmt. Wenn der Wartungsaufwand zu hoch wird, muß das System durch etwas anderes ersetzt werden.

Obwohl die Wartung einen erheblichen Anteil der eingesetzten Ressourcen erfordert, ist ihr Status sowohl in der Wissenschaft, als auch in der Praxis nicht z. B. dem Softwaredesign ebenbürtig (siehe z. B. [Mue97, S. 5 und S. 8 f.]). Kroha weist auch auf das geringe Ansehen der Wartung beim Wartungspersonal hin.[1] In [KGa95] ist nachzulesen: „Die Wartung von Software wurde lange Zeit als profane Aufgabe gesehen, die weder produktiv noch umsatzsteigernd ist. Wartung wurde demzufolge als notwendiges Übel betrachtet und in vielen Firmen den Junior Programmierern aufgebürdet ...“.[2]

Gründe für den relativ hohen Ressourcenbedarf liegen darin, daß ein Softwaresystem unter Umständen in relativ kurzer Zeit entwickelt wird, dann aber für einen potentiell sehr langen Zeitraum genutzt wird. Die Wartung muß während des Nutzungszeitraums geänderten und neuen Anforderungen Rechnung tragen (adaptive Wartung), Fehler beseitigen (korrektive Wartung), Systemei-

[1] [Kro97, S. 182]
[2] [KGa95, S. 3 f]

genschaften verbessern (perfektive Wartung) sowie künftige Veränderungen vorausschauend vorbereiten (präventive Wartung).[3]

Wartung verändert Softwaresysteme.[4] Die Veränderungen wirken sich meist negativ auf die Struktur der Software aus, sie wird immer komplexer: „*As a program evolves, it becomes more complex, and extra resources are needed to preserve and simplify its structure.*“[5] Erhöht sich die Komplexität einer Software, dann wird sie schwerer durchschaubar und testbar. Es treten potentiell mehr Fehler auf und die Software wird insgesamt schwerer wartbar. Es müssen also zusätzliche Anstrengungen unternommen werden, um dem entgegenzuwirken. Von den genannten Wartungsarten dient vor allem die präventive Wartung, teilweise aber auch die perfektive Wartung diesem Zweck.

Vor welchen Problemen steht jedoch das Wartungspersonal? Zum einen sind alte und große Systeme zu warten, die für das Funktionieren und Überleben von Organisationen unverzichtbar sind. Oft enthalten sie verstecktes Wissen z. B. über Geschäftsprozesse und können nicht einfach ersetzt werden. Diese Systeme werden als Altsysteme (bzw. *legacy systems*) bezeichnet und es gibt Beschreibungen für viele ihrer Probleme (siehe z. B. [Kro97, Kapitel 12]) sowie Vorschläge zu deren Lösung. Andererseits sorgen kurze Produktzyklen und sich schnell ändernde Umgebungsbedingungen (neue Gesetze, Firmenstrukturen, *Buzzwords* usw.) dafür, daß auch relative junge Systeme inflexibel und schwer wartbar werden.

Modifikationen und Anpassungen von Software im Rahmen der Wartung laufen meist ereignisgesteuert ab ([Kro97, S. 182]). Es tritt z. B. ein Fehler auf, der unter Zeitdruck beseitigt werden muß. Dabei wird unter Umständen nicht nur die Struktur der Software verschlechtert, sondern es wird auch versäumt, vorgenommene Änderungen zu dokumentieren. Hinzu kommt Personalfluktuation. Im Laufe der Zeit wird das Wissen über das System immer geringer, die Wartung teurer und langwieriger. Im schlimmsten Fall steht ein Wartungsprogrammierer unter Zeit- und Erfolgsdruck vor einem unbekannten, sehr großen System ohne (korrekter) Dokumentation. *Reengineering* verspricht, aus diesem Zustand herauszuhelfen.

Definitionen für *Reengineering* werden von vielen Autoren genannt (siehe z. B. [Mue97, S. 10 f]). Zitiert werden soll an dieser Stelle die Definition von Chikofsky und Cross:

> **Reengineering (Chikofsky und Cross)**: Untersuchung und Modifikation eines Programmsystems, um es in einer neuen Form wiederherzustellen und diese Form nachfolgend zu implementieren.[6]

Anhand der Definition wird deutlich, daß es hierbei mindestens um die Analyse eines vorhandenen Systems und dessen Modifikation geht. Es wird betont, daß das Endergebnis wiederum ein implementiertes Programmsystem ist.

Die untersuchende und analysierende Komponente des *Reengineering* wird mit dem ähnlich klingenden Begriff *Reverse Engineering* benannt.

---

[3] [Kro97, S. 181 f]

[4] Zitat von C. McClure: „*Software maintenance is the process of changing software systems.*“. Übernommen aus [Mue97, S. 8].

[5] Bekannt als *Law of increasing complexity*, aufgestellt von Lehman/Belady. Übernommen aus [DDN03, S. xvii]

[6] Zitiert aus [Mue97, S. 10]. Zu finden in ähnlicher Form auch in anderen Werken, z. B. in [KGa95, S. 9] oder [DDN03, S. 5]

> **Reverse Engineering**: Unter *Reverse Engineering* versteht man die Extraktion und Repräsentation von Informationen aus einem Software-System (Spezifikation) in einer anderen Form oder auf einem höheren Abstraktionsniveau.[7]

Diese Definition ist deutlich als Gegenstück zum *Forward Engineering* formuliert, welches den umgekehrten Weg beschreibt:

> **Forward Engineering**: *Forward Engineering* ist der Transformationsprozeß einer Spezifikation von einem höheren in ein niedrigeres Abstraktionsniveau.[8]

Ein wichtiges Stichwort aus dem Bereich des *Reverse Engineering* ist *Design Recovery*. Dieser Begriff drückt eigentlich aus, was intuitiv mit *Reverse Engineering* in Verbindung gebracht wird: das Aufdecken dessen, was Designer und Programmierer dazu bewegt hat, die vorliegende Architektur und Implementierung zu schaffen.

> **Design Recovery**: Im *Design Recovery* erstellt man ein Modell des betrachteten Systems (Programm). Dieses Modell besitzt ein höheres Abstraktionsniveau und kann nicht ausschließlich aus dem System (Programm) erzeugt werden. ...[9]

Im Gegensatz zum Design während des *Forward Engineering* ist hier in erster Linie also ein bestehendes System Gegenstand der (Re-)Modellierung und nicht ein Problembereich. Fehlende oder unklare Informationen müssen eventuell anders beschafft werden, möglicherweise aber auch auf die gleiche Art wie im *Forward Engineering*.

Ein Softwaresystem besteht idealerweise nicht nur aus Quellcode, sondern aus einer Vielzahl weiterer Dokumente (Artefakte), die unterschiedliche Gesichtspunkte des Systems auf unterschiedlichen Abstraktionsniveaus beschreiben. Um diesem Idealzustand näher zu kommen sind weitere Anstrengungen nötig. Für *Reverse Engineering* wird als weitere wichtige Tätigkeit das Redokumentieren genannt.[10]

Wie in der Definition zum Ausdruck kommt, wird ein Softwaresystem während des *Reengineering*s verändert, modifiziert. Diese Veränderungen können in ihrer Art und in ihrem Umfang sehr unterschiedlich ausfallen. Bereits die bloße Formatierung des Quelltextes mit einem automatischen Formatierer kann eine Verbesserung darstellen. In älterer Literatur wird dagegen als Beispiel das Umwandeln von Programmcode gemäß den Regeln der Strukturierten Programmierung genannt.[11] Vorgänge solcher Art werden unter dem Oberbegriff *Restructuring* genannt:

> **Restrukturierung (Restructuring)**: Unter Restrukturierung versteht man die Transformation zwischen Repräsentationsformalismen ohne Änderung der Funktionalität (von außen beobachtbares Verhalten).[12]

[7] [Mue97, S. 11]
[8] [Mue97, S. 11]
[9] [Mue97, S. 12]
[10] Siehe z. B. [Mue97, S. 12] und [KGa95, S. 8 f]
[11] Siehe z. B. [Mue97, Kapitel 6].
[12] [Mue97, S 12]

Wie in [Mue97, S. 12 f] weiter beschrieben wird, können viele Restrukturierungsmaßnahmen „... ohne zusätzliches Wissen, wie Einsatz des Programms oder Problemwissen, erfolgen.“ Dies betrifft allerdings Transformationen rein syntaktischer Art.

Speziellen Arten von Problemen mit Altsystemen widmen sich andere Gebiete: z. B. Trennung von monolithischen Systemen in Module (Modularisierung), Umstieg auf eine neuere Version einer Programmiersprache oder auf eine andere Programmiersprache (Sprachenkonversion) oder Wiederverwendung (*Reuse*).

Für das *Reengineering* objektorientierter Systeme wurde in den letzten Jahren ein anderes Stichwort populär: *Refactoring*. Bekannt geworden ist dieser Begriff vor allem durch das gleichnamige Buch von Fowler ([Fow99]). Ein Hauptanliegen ist hier, die Codequalität durch Auffinden und Verbessern „gern gemachter“ Design- und Programmierfehler zu verbessern.

Die von Fowler stammende Definition des *Refactoring* ist der oben genannten *Restructuring*-Definition sehr ähnlich, wenn man einmal von der zusätzlichen Nennung der Ziele bei Fowler absieht:

> ***Refactoring*** *...: a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.*[13]

Dies läßt den Schluß zu, daß es sich dabei um eng verwandte Gebiete handelt (*Refactoring* als Spezialfall des *Restructuring*).[14]

Zusätzlich zu den genannten gibt es noch eine Vielzahl weiterer *Re*-Wörter: *Redesign*, *Renovation*, *Redevelopment* usw. Sie bilden jeweils Oberbegriffe für Teilgebiete des *Reengineering*, für spezielle Methoden von CASE-Werkzeugen oder sind schlicht veraltet (*Renovation*).

2.2 Einige Prozesse im Reengineering

Dieser Abschnitt widmet sich einigen Tätigkeiten, die beim *Reengineering* eines Softwaresystems eine Rolle spielen. Es soll nicht versucht werden, den vorhandenen Übersichten zu diesem Themengebiet eine weitere hinzuzufügen. Statt dessen werden einige Prozesse hervorgehoben. Deren Auswahl richtet sich in erster Linie nach den erwarteten Erfordernissen dieser Diplomarbeit.

Ausführlichere und tiefer gehende Informationen gibt es z. B. in [DDN03], [FAM99], [Kau94], [Kro97], [Mue97]. In den folgenden Abschnitten wird jeweils auf weiterführende Quellen verwiesen.

2.2.1 Programmverstehen

Programmverstehen ist der aktive Vorgang des Verstehens von Struktur, Eigenschaften und interner, technischer Arbeitsweise von Software.[15]. Es ist der erste, aktive Schritt beim Lösen von Wartungsaufgaben, besonders, wenn unbekannte Systeme betroffen sind oder unerfahrene Wartungskräfte ihre Arbeit aufnehmen. Ziel des Programmverstehens ist, daß ein Mensch mit einem bisher unbekannten System bekannt und vertraut ist.

[13] [Fow99, S. 53]
[14] In [DDN03, S. 9] findet sich der Satz: „*Refactoring is restructuring within an object-oriented context.*“
[15] [Mue97, S. 13 und 27]

In einer älteren, aber oft zitierten Studie von Fjeldstad und Hamlen ([FH79]) wird berichtet, daß das Verstehen von Anfrage und Kontext einer Wartungsaufgabe (Dokumentation, Programmcode) rund die Hälfte des Aufwandes für Wartungsaktivitäten ausmacht.[16] Es wird deutlich, „... daß eine Effizienzsteigerung des Programmverstehensprozesses die größten Auswirkungen zur Reduzierung der Software-Wartung haben kann.“[17]

Für das Programmverstehen existieren drei favorisierte Theorien:[18]

- **Bottom-up**: Ausgehend von Quellcode und Codedokumentation werden gefundene Zusammenhänge mit Mitteln höherer Abstraktionsstufen beschrieben. Dieser Vorgang beginnt bei kleinen Einheiten (z. B. Methoden) und erstreckt sich nach und nach auf das gesamte System. In [Mue97, S. 28 f] wird als Beispiel beschrieben, wie einem Stück Quellcode durch Untersuchung eine Bedeutung zugewiesen wird. Die Erkenntnis, daß das Stück Code eine Sortierung durchführt (und zwar im Beispiel *Bubble-Sort*), weist dem Code eine abstraktere Bedeutung zu. Vorausgesetzt, der Programmierer weiß mit dem Begriff *Bubble-Sort* etwas anzufangen, ist durch die Abstraktion ein einfacheres und daher potentiell schnelleres Verstehen möglich.

- **Top-down**: Ausgehend von Wissen über den Problembereich (Anwendungsdomäne) werden Hypothesen über das Vorhandensein von Bestandteilen des Systems aufgestellt und überprüft. So wird man bei einer Personalverwaltung Angaben zu Personen in bestimmten Datenstrukturen vermuten (z. B. Records oder Klassen). Ausführliche Beispiele hierzu finden sich in [DDN03, S. 74 ff und S. 80 ff].

- **Kombiniert**: Die beiden zuerst genannten Verfahren werden abwechselnd verwendet. Die Entscheidung trifft der Analysierende anhand seiner Einschätzung der Erfolgswahrscheinlichkeit und Effizienz. In [Mue97, S. 30] wird dieses Vorgehen auch „opportunistisch“ genannt. Ein Verfahren, daß diesen Ansatz folgt, wird z. B. in [Rug92] beschrieben.

Als „verstanden“ kann ein System, bzw. ein Teil eines Systems, dann gelten, wenn es erklärt werden kann. Es versteht sich von selbst, daß gewonnene Erkenntnisse sofort dokumentiert werden müssen, wenn die geleistete Arbeit einen nachhaltigen Wert haben soll. Erkenntnisse gehen so nicht verloren und können „wiederverwendet“ werden, d. h. der Erkenntnisprozeß verkürzt sich bei anderen Beteiligten. Fehlerhafte Erkenntnisse können später nachvollzogen werden. Der Zwang zur Beschreibung dessen, was man vorher nur gedacht hat, führt nach eigener Erfahrung außerdem zu einer Aufbereitung, „Konsistenzprüfung“ und besseren Erinnerbarkeit des gewonnenen Wissens.

Verschiedene negative Einflüsse beeinträchtigen das Verstehen und können zu Fehlern führen. Dazu gehören einerseits unbekannte und ungewohnte Programmiersprachen oder Technologien, Programmiertechniken (z. B. sogenannte geniale Programmiertricks), Zeit- und Erfolgsdruck und anderes. Negativ kann sich schon eine ungewohnte Formatierung des Quelltextes oder eine unverständliche Namenswahl für Bezeichner auswirken.

[16] Es werden genannt: 47 % des Aufwandes für Erweiterungen und 62 % für Korrekturen.
[17] [Mue97, S. 27]
[18] [Mue97, S. 29], [Rug92]

Neben der Dokumentation (sofern vorhanden und konsistent) können weitere Faktoren positiv auf den Prozeß des Verstehens einwirken. Dazu gehören geeignete Werkzeuge (siehe Abschnitt 2.3). Diese gestatten z. B. die Visualisierung und graphische Aufbereitung von Quellcode, oder sie ermöglichen Navigation im Quellcode, Suche und Anfragen usw.

2.2.2 Messen

Einige Eigenschaften von Software lassen sich durch Messen ermitteln, andere Eigenschaften können von gemessenen Größen abgeleitet oder abgeschätzt werden. Metriken werden an vielen Stellen in der Softwaretechnik eingesetzt, z. B. um den Softwareentwicklungsprozeß organisatorisch und betriebswirtschaftlich beherrschen zu können.

Man unterscheidet zwischen direkten Metriken, die im Gegensatz zu indirekten Metriken nicht von anderen Messungen abhängig sind. Direkte Metriken sind in der Regel einfach zu bestimmen. Man kann Metriken anhand der betrachteten Größen unterscheiden. Metriken interner Größen betrachten die Software aus dem Blickwinkel des Programmierers, Metriken externer Größen dagegen aus der Sicht des Anwenders.[19]

Beispiele für Metriken externer Größen sind die *Function-Point*- und die *Feature-Point*-Metriken.[20] Mit ihrer Hilfe kann man zum Zeitpunkt der Anforderungsspezifikation den Aufwand für die Entwicklung abschätzen. Sie beruhen sehr stark auf Abschätzung der funktionalen Eigenschaften eines meist noch nicht existierenden Softwaresystems und sind damit stark von Problemanalyse und den Fähigkeiten des Schätzers abhängig. Laut [Mue97, S. 57] ist bei einer Aufwandsabschätzung mittels *Function-Points* immer die gesamte Anwendung zu betrachten. Aus diesem Grund scheint diese Metrik nicht geeignet für Wartungsprojekte zu sein, die sich typischerweise mit Teilen von Anwendungen beschäftigen. Die im weiteren Verlauf dieses Abschnitts genannten Metriken beziehen sich sämtlich auf interner Größen.

In der Wartung und speziell beim Reengineering werden Metriken benötigt, um die Ergebnisse der Arbeit einschätzen zu können. Das betrifft den qualitativen Zustand des Programmcodes und seine Wartbarkeit sowie die Veränderung dieser Eigenschaften infolge von Modifikationen. Leider können gerade diese Eigenschaften nicht (direkt) gemessen werden. Sie müssen von anderen Metriken abgeleitet werden.

Eine naheliegende und oft verwendete direkte Metrik ist die Größe eines Programms. Diese wird häufig einfach anhand der Zahl der Programmzeilen bestimmt. In der einfachsten Form werden alle Zeilen im Quelltext gezählt, die nicht leer sind und nicht nur aus Kommentaren bestehen. Die Maßeinheit ist LOC[21]. Sie wird häufig in Verbindung mit den Präfixen kilo (KLOC) oder Mega (MLOC) verwendet.

Es existieren aber auch andere Verfahren.[22] Der Vorteil alternativer Verfahren liegt oft in einer von der verwendeten Programmiersprache und dem Codestil

[19] [Kro97, S. 221 f]
[20] Siehe z. B. [Kro97, S. 224 ff] oder [Bal00, S. 83 ff]
[21] Lines of Code
[22] [Kro97, S. 232 f] oder [Mue97, Abschnitt 4.1.2] beschreiben z. B. die Metrik der Größe eines Programms nach Halstead. Diese wird nicht anhand der Zahl der Zeilen eines Programms, sondern anhand von syntaktischen Einheiten (*Token*) bestimmt.

unabhängigeren Messung.[23] Den Angaben zu LOC wird dann häufig noch ein Zusatz angehängt, um auf den Unterschied hinzuweisen (z. B. LOC NCSS[24])

Laut [Mue97, S. 48] kann allerdings ein linearer Zusammenhang zwischen verschiedenen Metriken zur Programmgröße angenommen werden. Unter dieser Annahme würde die einfache Messung der Programmzeilen ausreichen, um zumindest Vergleiche zwischen Systemen zu ermöglichen, die mit ähnlichen Programmiersprachen und Technologien implementiert sind.

Kritik an umfangsorientierten Metriken findet man z. B. in [Kro97, S. 230 ff]. Zusammenfassend kann man sagen, daß diese Metriken (wie eigentlich alle Metriken) nicht zur uninterpretierten Verwendung taugen, etwa um Kosten oder Entlohnung direkt damit zu verknüpfen. So sagt die Anzahl der Codezeilen nichts über die Komplexität des Quellcodes und wenig über den Aufwand zu dessen Erstellung aus.

Praktisch werden umfangsorientierte Metriken dennoch oft zur Angabe der Größe von Systemen verwendet (im Sinne von je größer, desto mehr Aufwand oder desto komplexer). Man findet auch Angaben, die die Leistungsfähigkeit von Werkzeugen oder das geleistete Arbeitspensum bewerten sollen. Außerdem dienen umfangsorientierte Metriken auch als Basis zur Ableitung anderer Metriken.[25]

Der Frage der Komplexität widmet sich die Metrik der zyklomatischen Komplexität von McCabe. Diese Metrik bestimmt die Anzahl unabhängiger Ablaufpfade am Ablaufplan eines Programms (bzw. an Teilen von Programmen, wie Prozeduren oder Methoden). Informationen zum graphentheoretischen Hintergrund und zur Berechnung befinden sich z. B. in [Kro97, S. 233 ff] und [Mue97, S. 50 ff].

McCabes zyklomatische Komplexität stützt sich auf die Erkenntnis, daß die Komplexität eines Programms von der Anzahl unabhängiger Ablaufpfade abhängig ist. Eine hohe Zahl von Pfaden bedeutet eine komplexe Ablauflogik, die schwerer zu verstehen und zu testen ist. So ist pro Pfad mindestens ein Testfall nötig. In [Kro97, S. 233] wird ein Wert von 10 als ungefährer Grenzwert für wesentlich erschwerte Wartung genannt.

Die McCabe-Metrik bewertet alle unabhängigen Ablaufpfade gleich. Eine Verfeinerung kann darin bestehen, die verschiedenen Mittel der Programmiersprache für Verzweigungen unterschiedlich zu bewerten.[26] Eine „gewünschte“ Verzweigung mit `if-else if-else` kann leichter zu interpretieren sein, als eine implizite, vielleicht nicht einmal erkannte Verzweigung in einem `try-catch`-Block. Wie sich im Abschnitt 2.3.2 zeigen wird, treffen hier Werkzeuge für Metriken auch unterschiedliche Entscheidungen.

Mit der Betrachtung von Beziehungen zwischen Modulen beschäftigt sich die Henry-Kafura-Metrik. Auch diese Metrik hat einen Bezug zu Graphen, denn die Beziehungen werden als Graph dargestellt. Hierbei bilden Datenstrukturen oder Unterprogramme die Knoten und Datenflüsse die Kanten. Betrachtet werden für

[23] Meßprogramme für die Programmgröße definieren häufig genaue Regeln, was wirklich als Codezeile gezählt wird, um solche Einflüsse zu minimieren. Ein Beispiel einer solchen Definition ist in der Hilfe zu JavaNCSS zu finden (zu JavaNCSS siehe auch Abschnitt 2.3.2).

[24] Non-Commented Source Statement

[25] [Kro97, S. 232] enthält eine kurze Übersicht.

[26] [Kro97, S. 234]

einen Knoten die Zahl der eingehenden Kanten (*fan-in*) und der ausgehenden Kanten (*fan-out*) sowie die Komplexität des bearbeitenden Programmcodes. Aus diesen Werten läßt sich eine Maßzahl für die Komplexität eines Moduls berechnen.[27]

Einige Metriken aus der objektorientierten Welt beschäftigen sich auch mit Modulen sowie mit Abschätzungen von deren Qualität. Ein Beispiel sind die in [Mar95] beschriebenen Modulabhängigkeiten. Das Werkzeug JDepend, welches in dieser Arbeit eingesetzt wird, implementiert einige dieser Metriken.[28]

Zur Betrachtung der Eigenschaften objektorientierter Software gibt es ebenfalls spezielle Metriken. Einige Beispiele für bekannte objektorientierte Metriken sind:

- Anzahl der Attribute, Anzahl der Methoden (**Number of Methods**, NOM) (mit oder ohne Einbeziehung von Attributen und Methoden mit Zugehörigkeit zu Klassen)
- Tiefe des Vererbungsbaums (**Depth of Inheritance Tree**, DIT), Anzahl der Kinder, d. h. Subklassen einer Klasse (**Number of Children**, NOC)
- **Number of Overriden Methods** (NORM): Anzahl überschriebener Methoden
- **Weighted Method Count** (WMC): Summe der Komplexitäten für alle Methoden einer Klasse; Die Komplexitäten können als konstant angenommen werden (mit 1), wobei die Metrik dann zu NOM mutiert. Die Komplexität einer Methode kann aber z. B. mit der McCabe-Komplexität berechnet werden.[29]
- **Lack of Cohesion** (LCOM): Angabe einer Maßzahl für die Kohäsion innerhalb einer Klasse. Für LCOM existieren verschiedene Berechnungsvorschriften.[30] Das ab Seite 25 vorgestellte Werkzeug net.sourceforge.Metrics benutzt z. B. die Berechnungsmethode nach Henderson-Sellers und liefert Werte im Intervall [0, 1].[31]

Es ist klar, daß sich diese Metriken auf spezielle Eigenschaften objektorientierter Programme richten und zum Teil sogar unabhängig von der verwendeten Programmiersprache sind. Einige weitere objektorientierte Metriken sind in [Kro97, S. 235 ff] beschrieben. Objektorientierten Metriken, Anhaltspunkten zu ihrer Interpretation und Verweisen zu Kritik ist in [FAM99] Abschnitt 20.3 gewidmet.

Auch im Fall objektorientierter Metriken bleibt die Frage nach der Interpretation der gemessenen oder erhaltenen Werte bestehen. Im Bereich dieser Arbeit ist die Frage wichtig, wie man Wartbarkeit und Codequalität bestimmen kann. Einer großen Zahl an Metriken steht eine relativ kleine Zahl an bekannt gewordenen Untersuchungen und Evaluierungen gegenüber.[32] Die Abschnitte

[27] [Kro97, S. 229 f]
[28] Siehe Abschnitt 2.3 zu JDepend.
[29] [FAM99, S. 270 f]
[30] [FAM99, S. 278 ff]
[31] Siehe die Hilfe zum Werkzeug net.sourceforge.Metrics.
[32] [FAM99, S. 287]

20.4 und 20.5 in [FAM99] enthalten Fallstudien, in denen unterschiedliche objektorientierte Metriken an realen Softwaresystemen angewandt und überprüft wurden.

Ein interessantes Ergebnis dieser (und auch anderer) Studien ist, daß sich in Projekten verschiedene Meßwerte für die meisten Klassen in einem bestimmten Rahmen bewegen. Lediglich einige „Ausreißer“ brechen aus diesem Rahmen aus. Auf diesen Umstand weist auch [DDN03, Pattern 4.3 auf S. 84 ff] hin und empfiehlt, bei den „Ausreißern“ mit der Suche nach Designanomalien zu beginnen. Eingegangen wird z. B. auf den möglichen Zusammenhang zwischen Größe einer Klasse (bestimmbar an der außerordentlich großen Zahl an Methoden, Attributen aber auch NCSS) und Designfehlern (zu viele Verantwortlichkeiten der Klasse). Es wird jedoch auch darauf hingewiesen, daß bisher keine Methode (z. B. Schwellwerte für Messungen) bekannt ist, anhand deren man Fehler im Design automatisch bestimmen könnte.

2.2.3 Restrukturierung — Refactoring

Restrukturierung ist die Umwandlung einer Programmbeschreibung (Quellcode) von einer Form in eine andere, ohne das von außen beobachtbare Verhalten des Programms zu verändern (siehe Definition auf Seite 12).

Ziel der Restrukturierung ist es, die Qualität des Designs und des Programmcodes eines Softwaresystems zu verbessern, insgesamt also den Aufwand für die Wartung zu reduzieren. Es ist hierbei zu bedenken, daß Restrukturierung im Rahmen von *Reengineering* ausgeführt wird. Ausgangspunkt ist also ein System mit bereits hohem Alter und/oder hohem Modifikationsgrad.

Zu Bedeutung kam der Begriff *Restructuring* bereits in Verbindung mit der Strukturierten Programmierung. Hier bestand die Herausforderung darin, Programmcode von unerwünschten Konstrukten zu befreien (z. B. extensive Verwendung von `GOTO`'s, die zu Spaghetticode führt[33]) und insgesamt durch eine kleine Menge standardisierter Programmelemente und -strukturen zu ersetzen.[34]

Eine weitere Tätigkeit ist das Einführen oder Redefinieren von Modulen, die Modularisierung. Eingesetzt werden sollte Modularisierung vor allem, um große, monolithische Altsysteme in kleinere, weniger komplexe und damit potentiell besser wartbare Einheiten aufzuteilen.

Die Aufteilung eines Systems in Module verringert die Komplexität innerhalb der gebildeten Module (gemessen an der Komplexität des vorherigen Monolithen). Allerdings ist auch die Komplexität der Kommunikation zwischen den Modulen zu berücksichtigen. Diese nimmt zu, je mehr Module gebildet werden. Das Ziel einer erfolgreichen Modularisierung ist es, ein Optimum dieser gegenläufigen Trends zu finden.[35]

Zur Verringerung der Komplexität der Kommunikation zwischen Modulen gibt es zwei Maßnahmen: die Verringerung des Kontrollflusses und die Verringerung des Datenflusses zwischen den Modulen. Daten sollen dabei, wenn möglich, im Modul verarbeitet und nicht zwischen Modulen transportiert werden. Die Anzahl und Art der Moduldienste beeinflußt die Komplexität des Kontrollflusses.

[33] Nach Andrew S. Tanenbaum könnte eine Lösung darin bestehen, `GOTO` umzubenennen. Er schlug als Alternative `IKNOWTHISISASTUPIDTHINGTODOBUTNEVERTHELESSGOTO` vor. Im Zeitalter von Programmierumgebungen mit Vervollständigenfunktion hilft dies allerdings wenig.

[34] Siehe z. B. [Kau94, S. 35 ff].

[35] [Kro97, S. 108 f]

Der Begriff der Kohäsion[36] bezeichnet, wie eng Dienste eines Moduls semantisch zusammengehören. Angestrebt wird eine hohe Kohäsion.[37]

In der objektorientierten Welt ist seit einiger Zeit das bereits erwähnte Schlagwort *Refactoring* populär. Zumindest eine Wurzel liegt in der Smalltalk-Welt. Für diese Programmiersprache gibt es auch ein bekanntes Werkzeug, den *Refactoring Browser*, der diese Art von Codemanipulation für den Programmierer vereinfacht. *Refactoring* wird auch als ein wichtiges Element im *Extreme Programming* genannt, was jedoch außerhalb dieser Arbeit liegt.[38]

Der Kern des *Refactoring* ist eine Menge von Regeln (wie beschrieben in [Fow99]) oder *Reengineering*-Muster (wie teilweise in [DDN03]). *Refactoring* beschreibt die Modifikation von bestehender Software in kleinen Schritten und von systematischen Tests begleitet. Fowler nennt als Ziele:

- Die Verbesserung der Lesbarkeit und Verständlichkeit durch Verringerung von Komplexität in bestimmten beschriebenen Fällen.
- Die Verbesserung der Wartbarkeit, durch weniger komplexe und verständlichere Programme.
- Die Verbesserung der Erweiterbarkeit durch Befolgen bestimmter Designregeln.

Es wird prinzipiell davon ausgegangen, daß durch Erweiterungen komplexerer und schlechterer Code entsteht. Solchen Qualitätsverschlechterungen soll durch diszipliniertes Vorgehen und Umgestalten in kleinen Schritten begegnet werden.

Auch *Refactoring* bietet keine automatisierte Methode, um Codequalität zu verbessern. Das Erkennen von Problemen und die Entscheidung für eine gute Lösung muß auch hier der Programmierer[39] treffen. Die in Form von Regeln abgefaßten *Refactorings* beschreiben jedoch Indizien, an denen man ein Problem erkennen kann, nennen mögliche Lösungen und Alternativen. Es gibt aber auch Veröffentlichungen, die sich mit der Unterstützung des *Refactoring* durch Metriken beschäftigen (z. B. [Mar95] oder [SSL01]).

Während *Refactoring* sich in erster Linie auf Transformationen von Programmcode bezieht, zielen die sogenannten *Reengineering patterns* auf Vorgänge während des gesamten Prozesses des *Reengineering*. Ein Teil bezieht sich dabei auch auf Maßnahmen zur Restrukturierung von problematischem Programmcode. Die Muster sind analog zu *Design patterns* formuliert. Veröffentlicht wurden dazu z. B. [DDN03] und [FAM99].

2.2.4 Wiederverwendung (Reuse) von Programmcode

Die Wiederverwendung von Ergebnissen anderer oder eigener Arbeit kann ein wichtiges Mittel zur Produktivitätssteigerung sein. Man muß nicht jedes mal das Rad neu erfinden, um ein neues Fahrzeug zu bauen. Wiederverwendung

[36] Im vorigen Abschnitt wurde bereits eine Metrik für diese Eigenschaft bezogen auf Klassen beschrieben. LCOM mißt allerdings das „Fehlen“ der Kohäsion.

[37] [Kro97, S. 110 f]

[38] Zur Entstehung des Begriffes *Refactoring* siehe [Fow99, S. 71 f]. Ein wichtiges Buch zum Prozeß des *Extreme Programming* ist Beck, Kent: *eXtreme Programming eXplained: Embrace Change*. Reading, MA.: Addison-Wesley, 2000.

[39] Hier ist tatsächlich der (Wartungs-)Programmierer gemeint, nicht etwa ein Designer.

ist mit Standardisierung (auch in kleinem Maßstab) verbunden. Während ein Fahrzeugbauer jedoch nicht die Schrauben für jedes neue Fahrzeug neu erfindet, ist ein solches Vorgehen in Softwareprojekten nicht unüblich.

Wiederverwendung bezieht sich auf alle Ergebnisse der Softwareentwicklung, nicht nur auf Programmcode. Bemühungen, dies zu erreichen, gibt es daher auch für allen Phasen der Softwareentwicklung. Ein Beispiel für die Designphase sind die *Design patterns* der *Gang of Four* ([GoF96]) und anderer Literatur aus diesem Bereich (z. B. [Bus96]). *Design reuse* wird das größte Potential für die Steigerung der Produktivität und die Verringerung der Kosten zugesprochen.[40]

Bedeutsam, vor allem im *Reengineering*, ist jedoch die Wiederverwendung von Programmcode. Wiederverwendeter Code kann sehr unterschiedlichen Quellen entstammen:

- **Bibliotheken**: Als Ergebnis eigener oder fremder Entwicklungen entstehen Programmbibliotheken. Diese bieten in der Regel häufig benötigte, für Anwendungen allgemein wichtige oder schwierig zu entwickelnde Funktionen. Beispiele sind z. B. Bibliotheken mit oft benutzen Datenstrukturen (Package `java.util`), graphische Bibliotheken (Trolltech's Qt[41], Swing[42]) oder Bibliotheken für mathematische Probleme (diverse). Ein anderes prominentes Beispiel ist die *Standard Template Library* (STL) zu C++[43].

- **Codegeneratoren**: Wissen und Erfahrungen sowie Domänenmodelle fließen in Codegeneratoren zusammen. Diese werden in einer Generatorsprache programmiert und erzeugen daraus Programmcode in einer Zielsprache. Auf diese Weise werden Handlungen automatisiert, die sonst (wiederholt) manuell ausgeführt werden müssten. Beispiele sind Lex und Yacc, mit deren Hilfe Compiler generiert werden können, sowie graphische Werkzeuge zur Erzeugung von GUI[44]-Elementen, wie sie in vielen Programmierumgebungen mittlerweile enthalten sind.

- **Komponenten**: Als Komponenten werden Softwarebausteine bezeichnet, die einem definierten Programmiermodell gemäß erzeugt werden müssen. Sie verfügen in der Regel über einen Mechanismus zur Publizierung ihrer Eigenschaften (z. B. XML[45]-Beschreibungen) und dienen oft als Grundbausteine für „zusammenklickbare" Applikationen. Modelle dieser Art gibt es z. B. in Sprachen wie Visual Basic, Delphi, aber auch Java.

- **Frameworks**: In bestimmten Bereichen entwickelte Software enthält sehr häufig ähnliche Komponenten und hat eine ähnliche Struktur, unabhängig vom konkreten Leistungsumfang der einzelnen Applikationen. Für solche Anwendungsbereiche existieren häufig sogenannte Frameworks, die diese gemeinsame Struktur und Komponenten im Sinne eines Gerüstes vorgeben. Der Nutzer eines Frameworks muß im einfachsten Fall nur die Lücken mit seinem spezialisierten Code füllen sowie Erweiterungen anbauen. Ein Beispiel eines Frameworks für Webapplikationen ist Struts[46].

[40] [BM98, S. 37 f]
[41] http://www.trolltech.com
[42] http://java.sun.com/products/jfc/tsc/index.html
[43] z. B. http://www.sgi.com/tech/stl/
[44] Graphical User Interface
[45] Extensible Markup Language
[46] http://jakarta.apache.org/struts/

- **Altcode**: Programmteile aus einem anderen Projekt, die nicht explizit für die Wiederverwendung entworfen wurden, können ebenfalls in neue Projekte einfließen. Die Möglichkeiten reichen hier von der unmodifizierten Verwendung ganzer Systeme oder Subsysteme[47] bis hin zu *Copy-and-Paste* einzelner Methoden.

Diese Übersicht ist nicht vollständig und soll lediglich einige Beispiele anführen. Die einzelnen Punkte sind manchmal auch nicht strikt voneinander zu trennen.

Ein bisher nicht genannter Aspekt ist die Erkennung von gemeinsam benutzbaren Softwarekomponenten während Entwurf und auch *Reengineering*. Diese kommt in der Realität, besonders bei großen Projekten zu kurz. Müller nennt als Gründe dafür mangelnde Kommunikation zwischen verschiedenen Teams und das Fehlen einer organisationsweiten Infrastruktur speziell für Wiederverwendung.[48]

Gemeinsam ist den genannten Beispielen, daß sie im Idealfall Aufwand sparen und die Produktivität erhöhen können. Was man hat oder kauft, muß man nicht selbst entwickeln oder debuggen. Im schlimmsten Fall kann sich diese Annahme aber ins Gegenteil verkehren. Dies kann passieren, wenn der eigene oder gekaufte Code fehlerhaft ist oder andere Anforderungen nicht erfüllt. Katastrophal kann sich auch die Pleite eines Fremdherstellers auswirken, nämlich dann, wenn der Support wegfällt und wenn schlimmstenfalls die Investition (Geld, Einarbeitung, Projektzeit) verloren ist. Um dies zu vermeiden, müssen einige wichtige Eigenschaften der wiederverwendeten Software bekannt sein.[49]

An den aufgezählten Beispielen für die Wiederverwendung werden weitere Vorteile deutlich. So können Routinetätigkeiten entfallen (Nutzung von Bibliotheken, Codegeneratoren). Wiederverwendeter Code ist kein duplizierter Code und muß nur an einer Stelle in der Organisation gewartet und weiterentwickelt werden. Im Falle eines Fremdherstellers geht ein Teil des Risikos an eine andere Organisation über. Außerdem profitiert der Wiederverwender von Erfahrungen und sogar Spezialwissen anderer und kann die verfügbaren Ressourcen in für ihn wichtigere Belange investieren. Wird Software in vielen anderen Systemen verwendet, dann ist die Nutzerbasis breiter und die Qualität des Codes kann davon profitieren.[50]

Natürlich sind für diese Vorteile auch gegenteilige Szenarien denkbar, wie weiter oben bereits angedeutet. Problematisch kann auch die zusätzliche Kommunikation zwischen Entwicklern und Nutzern von wiederverwendbarem Code sein, z. B. wenn man für gekauften Code keinen ausreichenden Support erhält.

Wiederverwendbare Software oder Softwarekomponenten sind teurer und schwieriger zu entwickeln, als solche, die nur auf einen bestimmten Zweck ausgelegt werden. Vorteile der Wiederverwendung kommen erst bei späterer Nutzung zum Tragen und sind teilweise spekulativ (War die Entwicklung allgemein genug, um auch in ... Jahren noch von Nutzen zu sein?). Wird die geleistete Arbeit gar an den erzeugten LOC gemessen und entlohnt, schneidet die Entwicklung wiederverwendbarer Software schlechter ab und erntet damit Nachteile. Aus

[47] Möglicherweise werden die wiederverwendeten Teile dann von anderen Programmteilen durch Interfaces oder Wrapper getrennt, die auch eine Anpassung z. B. von einem prozeduralen Altsystem an ein objektorientiertes System leisten.

[48] [Mue97, S. 99]

[49] Eine Betrachtung des Qualitätsaspektes findet sich in [Mue97, S. 102 ff].

[50] [Mue97, S. 98]

diesen Gründen wird oft nicht auf die Entwicklung von wiederverwendbarer Software gesetzt. Ausnahmen sind natürlich Firmen, deren Geschäftsziele speziell auf deren Entwicklung und Verkauf ausgerichtet sind, z. B. Hersteller von Bibliotheken.[51]

2.3 Vorhandene Werkzeuge

Im Abschnitt 2.2 wurde auf einige Prozesse und Handlungen im Rahmen des *Reengineering* eingegangen. Es wurde dort angedeutet, daß deren Auswahl sich nach den Erfordernissen dieser Arbeit richtete. Unter der gleichen Voraussetzung sollen in diesem Abschnitt einige dazu passende Werkzeuge vorgestellt werden. Diese Werkzeuge sind unentbehrliche Hilfsmittel, sie helfen unnötige Fehler zu vermeiden und nehmen Routinearbeiten ab.

Bei der Auswahl berücksichtigt wurden auch nur Werkzeuge, die frei verfügbar (im Sinne freier Software) waren. Benutzt wurde außerdem auch das CASE-Werkzeug Rational Rose®.[52] Da zu diesem Werkzeug jedoch ausreichend Dokumentation und *Tutorials* verfügbar sind, wurde es in diesen Abschnitt nicht mit aufgenommen.

2.3.1 Programmverstehen und -modifizieren

Jeder Programmierer wird als Voraussetzung für seine Arbeit zunächst eine mehr oder weniger komfortable Entwicklungsumgebung (z. B. eine IDE[53]) für notwendig erklären.[54] Deren Fähigkeiten gehen mittlerweile weit über das Editieren, Formatieren und Hervorheben von Quelltext hinaus. Besonders im Hinblick auf die in den vorigen Abschnitten genannten Tätigkeiten ist die Auswahl einer geeigneten IDE wichtig. Folgende fortgeschrittene Fähigkeiten werden für nötig gehalten:

- **Projektimport/-export**: Die Fähigkeit, vorhandenen Quellcode mit unterschiedlichen Verzeichnisstrukturen zu importieren. Zusätzlich die Fähigkeit des Exports in zumindest einige oft benutzte Formate (Zip, JAR usw.).
- **Quellcodenavigation**: Die Fähigkeit, Definitionen, Deklarationen und Referenzen zu Ausdrücken im Quellcode zu finden und den Code anzeigen zu können. Diese Fähigkeit ist wichtig für das Verstehen von Abläufen im Programm, da diese am Quelltext nachvollzogen werden können.
- **Suchfunktionen**: Neben reiner Textsuche (mit Unterstützung für reguläre Ausdrücke) sollte die Suche nach Sprachelementen und deren Eigenschaften unterstützt werden. Dient ebenfalls dem Programmverständnis.

[51] [Mue97, S. 99 f]
[52] `http://www.rational.com`
[53] Integrated Development Environment
[54] Geschmäcker sind verschieden und Meinungen über ideale Voraussetzungen zum Programmieren auch. Für manchen Programmierer ist der `vi` und eine Sammlung Kommandozeilentools genau das Richtige. Auch eine solche Umgebung soll hier als Entwicklungsumgebung verstanden werden.

- **Generierung von Standardkonstrukten**: Generierung von Gerüsten für häufig benutzte Sprachkonstrukte, wie Klassen, Methoden, Codefragmente wie Schleifen, `try-catch`-Blöcke, Kommentare usw. Zusätzlich die Fähigkeit, den Vorrat an Standardkonstrukten selbst erweitern zu können. Ziel ist hier die Erhöhung der Produktivität des Programmierers bei gleichzeitiger Vermeidung von unnötigen Fehlern.
- **Erweiterbarkeit**: Die Fähigkeit, Module oder Fremdwerkzeuge integrieren zu können. Wichtig ist auch die Möglichkeit, in den *Build*-Prozeß eingreifen zu können.
- **Einfaches Refactoring**: Kontrolliertes Umbenennen und Verschieben von Konstrukten unter Berücksichtigung aller Referenzen. Solche Handlungen könnten auch mit den standardmäßig vorhandenen Ersetzenfunktionen durchgeführt werden. Hier besteht jedoch immer die Gefahr, etwas Falsches zu ersetzen und/oder eine Ersetzung zu vergessen. Mit derartigen Funktionen werden also unnötige Fehler vermieden (die korrekte Funktion immer vorausgesetzt).

Viele moderne IDEs bieten die genannten Fähigkeiten in verschiedener Qualität. Für diese Arbeit ausgewählt wurde die IDE Eclipse[55]. Ausschlaggebend waren vorhandene Erfahrungen des Autors mit dieser Software. Positiv beeinflußt wurde die Entscheidung aber auch durch die gute Erweiterbarkeit des Systems und die vielen vorhandenen Zusatzmodule. Als weiterer Pluspunkt sind die *Refactoring*-Fähigkeiten zu werten, die über das oben geforderte Maß deutlich hinausgehen. Negativ könnte das Fehlen eines GUI-Editors gewertet werden, er war jedoch für diese Arbeit von untergeordneter Bedeutung.

2.3.2 *Metriken und Visualisierung*

Werkzeuge zum Bestimmen von Metriken bringen häufig auch Möglichkeiten zur Visualisierung von Meßwerten und anderen Eigenschaften mit.[56]

Metriken und Messen sind kein Hauptgegenstand dieser Arbeit. Dennoch sollten einige Untersuchungen durchgeführt werden, um Eigenschaften von bestehendem und neuem Code einschätzen zu können. Zu diesem Zweck wurden einige Werkzeuge ausgesucht.

Aspect Mining Tool (AMT)

Zur Lösung eines komplexen Problems wird dieses oft in Teilprobleme oder Aspekte[57] zerlegt, die dann getrennt gelöst bzw. behandelt werden. Für dieses Prinzip gibt es den Begriff *Separation of concerns*. Eine Strukturierung und Modularisierung gilt als gut gelöst, wenn *concerns* in Modulen gekapselt behandelt

[55] http://www.eclipse.org/

[56] Nach dem Motto: „Ein Bild sagt mehr als tausend Worte.“

[57] Laut Duden bezeichnet das Wort Aspekt einen Gesichtspunkt, einen Blickwinkel auf einen Sachverhalt oder eine Sache. Leider „kollidiert“ das Wort Aspekt jedoch mit dem Aspektbegriff der Aspektorientierten Programmierung, wo damit ein Modularisierungskonzept bezeichnet wird. Da in dieser Arbeit dieses Gebiet nur sehr leicht gestreift wird, wird im Folgenden der Begriff Aspekt in seiner herkömmlichen, im deutschen Sprachgebrauch üblichen Bedeutung verwendet. Der Autor empfindet „Aspekt“ als gute Übertragung des englischen Begriffes *concern*, wie etwa im Schlagwort *Separation of Concerns*. Beide Begriffe werden in dieser Arbeit synonym verwendet.

werden. Leider ist mit den Modularisierungsmechanismen aktueller Programmiersprachen keine Separierung aller *concerns* möglich. Praktisch gibt es eine (oder auch gar keine) dominante Zerlegung, der sich andere *concerns* unterordnen müssen (sogenannte *hidden concerns*).[58]

Fortgeschrittene Techniken, wie die Aspektorientierte Programmierung versuchen, dieses Problem zu lösen. Zur Anwendung dieser Technik auf Altsoftware ist es wichtig, die verstreuten *concerns* im Code aufzufinden und zu erfassen. Um diese Aufgabe lösen zu können wurde das Werkzeug AMT[59] entwickelt.

Prinzip und Wirkungsweise von AMT sind in [HK01] beschrieben. Das Werkzeug ermöglicht eine textuelle und typbasierte Suche über den gesamten Quellcode eines Java-Projektes. Es kann *hidden concerns* visualisieren. AMT bietet keine automatische Suche nach *hidden concerns*[60], sondern unterstützt vielmehr den Analytiker bei dessen Suche.

Abbildung 2.1 zeigt beispielhaft die Visualisierung einer Suchanfrage.

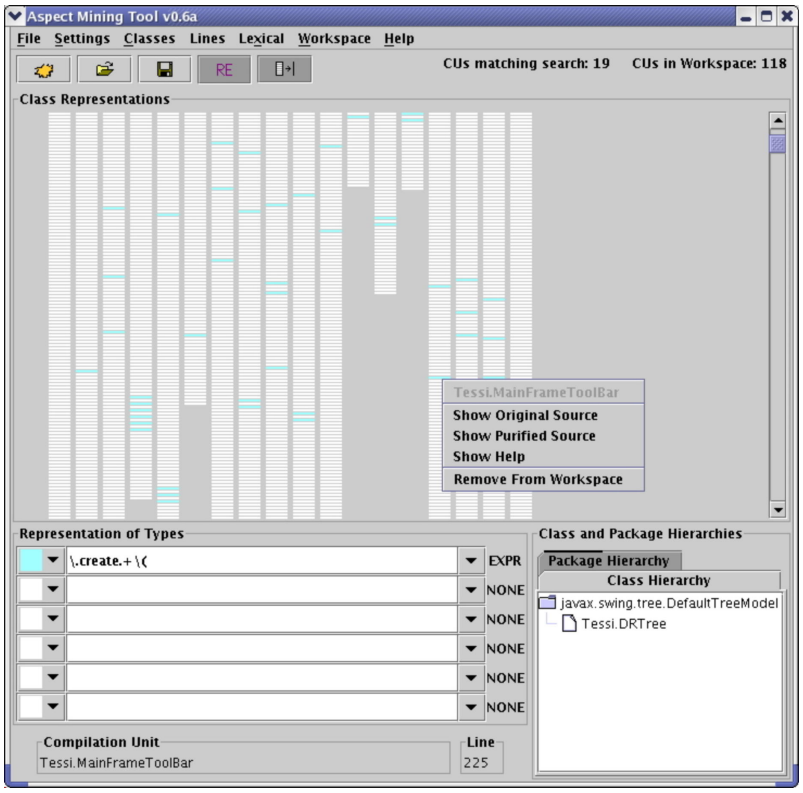

Abb. 2.1: Ansicht von AMT mit einer Beispielvisualisierung

In sechs Eingabezeilen kann eine Suche definiert werden. Zugelassen ist Suche auf Basis von Typen und Suchbegriffen. Es können auch reguläre Ausdrücke

[58] Siehe [HK01].

[59] http://www.cs.ubc.ca/~jan/amt/

[60] Es ist zweifelhaft, daß das vollautomatische Finden von *hidden concerns* in einem beliebigen Quelltext überhaupt möglich ist.

verwendet werden. Jedem der sechs Suchausdrücke kann eine Farbe zugewiesen werden.

Klassen in der Resultatmenge werden als Säulen dargestellt. Codezeilen mit Fundstellen der aktuellen Suche werden mit der gewählten Farbe hervorgehoben. Der Suchbereich und die Resultatmenge können auf der Basis von *compilation units* (also Java-Klassen) eingeschränkt werden. Zur genaueren Untersuchung von Fundstellen kann direkt an die gefundene Stelle im Code gesprungen werden.

JavaNCSS, JMetric, net.sourceforge.Metrics

Die drei Programme JavaNCSS[61], JMetric[62], net.sourceforge.Metrics[63] dienen der Berechnung von verschiedenen Metriken für Java-Sourcecode. Obwohl einige Metriken von allen drei Programmen ermittelt werden, ergaben sich dennoch Unterschiede für einzelne Werte. Die drei Werkzeuge wurden daher wechselseitig zur Prüfung von Werten benutzt.

JavaNCSS ist ein Java-Programm für die Konsole. Es ermittelt einige umfangsorientierte Metriken sowie McCabes zyklomatische Komplexität. Als Ergebnis erzeugt JavaNCSS einen Report des untersuchten Quelltextes in einem XML-Format. Zusätzlich ist ein XSL-Stylesheet zur Umwandlung in HTML vorhanden.

Gemessen wird im Einzelnen:

- Anzahl der Packages, Klassen, innere Klassen, Methoden
- Anzahl von NCSS pro Package, Klasse, Methode
- Anzahl von Javadoc-Kommentarzeilen
- Durchschnittswerte für Packages, Klassen, Methoden
- Zyklomatische Komplexität für Methoden

Im Abschnitt 2.2.2 wurde beschrieben, daß große Unterschiede in umfangsorientierten Metriken für eine Klasse im Vergleich zum Durchschnittswert des Projektes Indizien für Probleme sein können. JavaNCSS ist ein Werkzeug, um zunächst an diese Werte zu kommen.

Positiv an JavaNCSS ist noch zu erwähnen, daß das Programm problemlos in den *Build*-Vorgang mit Ant[64] integrierbar ist.

JMetric zeichnet sich durch eine graphische Darstellung der ermittelten Metriken aus. Das Werkzeug bietet eine Reihe von Diagrammen zu Metriken an.

Leider hat das Werkzeug einige Fehler, die den Einsatz für diese Arbeit einschränkten. Es wurde vor allem für Gegenprüfungen der Ergebnisse anderer Werkzeuge verwendet.

net.sourceforge.Metrics ist ein Eclipse-Plugin. Es ermittelt die größte Anzahl von Metriken. Der Hauptvorteil vor den beiden anderen Werkzeugen ist jedoch

[61] http://www.kclee.com/clemens/java/javancss/
[62] http://www.it.swin.edu.au/projects/jmetric/products/jmetric/
[63] http://www.sourceforge.net/projects/metrics
[64] Ant ist ein Werkzeug zur Steuerung von *Build*-Prozessen (ähnlich dem bekannten `make`). Die Homepage des Ant-Projektes ist http://jakarta.apache.org/ant/

die Integration in die IDE und die vielseitige und umfangreiche Ausgabe gemessener Werte in einer eigenen Ansicht. Darin werden jeweils Metriken angezeigt, wenn man ein Sprachkonstrukt (Packages, Klassen, Methoden) markiert. Dazu kommt die Berechnung von minimalen, maximalen und Durchschnittswerten, wo dies Sinn hat.

net.sourceforge.Metrics berechnet ebenfalls die für JavaNCSS erwähnten Metriken. Zu den von diesem Werkzeug zusätzlich berechneten Metriken gehören:

- Anzahl der Methoden (NOM), Anzahl von Klassenmethoden
- Anzahl der Attribute, Anzahl von Klassenattributen
- Tiefe des Vererbungsbaums (DIT)
- Anzahl der Kinder
- *Lack of Cohesion* (LCOM*)

Darüber hinaus gibt es weitere Metriken, die jedoch nicht betrachtet wurden. Abbildung 2.2 zeigt das Metrics-Ansichtsfenster aus Eclipse mit Werten zu TESSI 1.1.

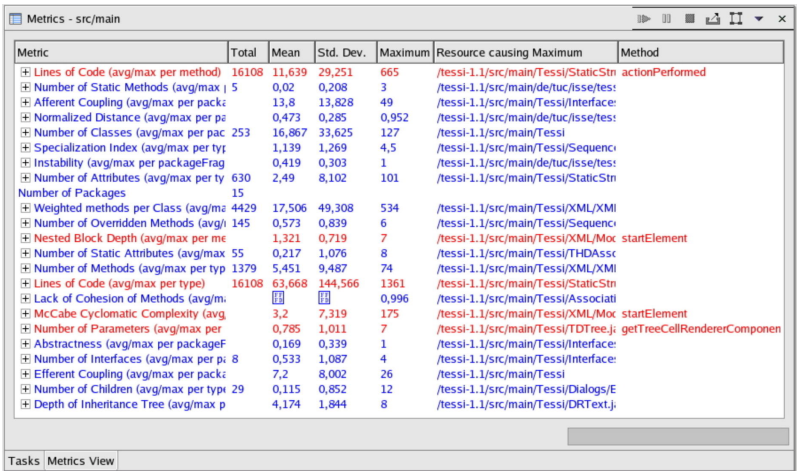
Metrics - src/main
Metric
Total
Mean
Std. Dev.
Maximum
Resource causing Maximum
Method
Lines of Code (avg/max per method) 16108 11,639 29,251 665 /tessi-1.1/src/main/Tessi/StaticStr actionPerformed
Number of Static Methods (avg/max 5 0,02 0,208 3 /tessi-1.1/src/main/de/tuc/isse/tes
Afferent Coupling (avg/max per packa 13,8 13,828 49 /tessi-1.1/src/main/Tessi/Interface
Normalized Distance (avg/max per pa 0,473 0,285 0,952 /tessi-1.1/src/main/de/tuc/isse/tes
Number of Classes (avg/max per pac 253 16,867 33,625 127 /tessi-1.1/src/main/Tessi
Specialization Index (avg/max per typ 1,139 1,269 4,5 /tessi-1.1/src/main/Tessi/Sequenc
Instability (avg/max per packageFrag 0,419 0,303 1 /tessi-1.1/src/main/de/tuc/isse/tes
Number of Attributes (avg/max per ty 630 2,49 8,102 101 /tessi-1.1/src/main/Tessi/StaticStr
Number of Packages 15
Weighted methods per Class (avg/ma 4429 17,506 49,308 534 /tessi-1.1/src/main/Tessi/XML/XMI
Number of Overridden Methods (avg/ 145 0,573 0,839 6 /tessi-1.1/src/main/Tessi/Sequenc
Nested Block Depth (avg/max per me 1,321 0,719 7 /tessi-1.1/src/main/Tessi/XML/Moc startElement
Number of Static Attributes (avg/max 55 0,217 1,076 8 /tessi-1.1/src/main/Tessi/THDAss
Number of Methods (avg/max per typ 1379 5,451 9,487 74 /tessi-1.1/src/main/Tessi/XML/XMI
Lines of Code (avg/max per type) 16108 63,668 144,566 1361 /tessi-1.1/src/main/Tessi/StaticStr
Lack of Cohesion of Methods (avg/m 0,996 /tessi-1.1/src/main/Tessi/Associati
McCabe Cyclomatic Complexity (avg 3,2 7,319 175 /tessi-1.1/src/main/Tessi/XML/Moc startElement
Number of Parameters (avg/max per 0,785 1,011 7 /tessi-1.1/src/main/Tessi/TDTree.j getTreeCellRendererComponen
Abstractness (avg/max per packageF 0,169 0,339 1 /tessi-1.1/src/main/Tessi/Interface
Number of Interfaces (avg/max per p 8 0,533 1,087 4 /tessi-1.1/src/main/Tessi/Interface
Efferent Coupling (avg/max per packa 7,2 8,002 26 /tessi-1.1/src/main/Tessi
Number of Children (avg/max per type 29 0,115 0,852 12 /tessi-1.1/src/main/Tessi/Dialogs/E
Depth of Inheritance Tree (avg/max p 4,174 1,844 8 /tessi-1.1/src/main/Tessi/DRText.j
Tasks
Metrics View

Abb. 2.2: Ansicht von net.sourceforge.Metrics in Eclipse. Zu sehen sind Metriken für TESSI 1.1

Bei der Evaluierung der Werkzeuge ergaben sich zum Teil erhebliche Unterschiede bei der Berechnung der McCabe-Komplexität. Bei genauerer Untersuchung ergaben sich Indizien, daß einige Sprachkonstrukte (z. B. `try-catch`-Blöcke) unterschiedlich bewertet wurden. Über Stichproben wurde ermittelt, daß net.sourceforge.Metrics nach Ansicht des Autors dieser Arbeit genau arbeitete. Es muß aber angemerkt werden, daß dies nur anhand einiger Beispiele geprüft wurde und nicht für alle Fälle bestätigt werden kann. Für kurze Methoden, in denen also potentiell nur wenige verschieden bewertete Sprachkonstrukte enthalten sind, unterschieden sich die Werte aller drei Werkzeuge wenig oder nicht. Benutzt wurde diese Metrik denn insgesamt auch nur qualitativ.

JDepend

JDepend[65] bestimmt Metriken, die Aussagen über die Qualität von Modulen ermöglichen sollen. JDepend implementiert einen Satz von Metriken, die in [Mar95] beschrieben sind.

JDepend ist für die Analyse von Java-Quellcode ausgelegt. Gemessen werden Werte bezogen auf Java-Packages. Betrachtet werden:

- Die Anzahl der eingehenden und ausgehenden Abhängigkeiten (Kopplungen, *couplings*) der Klassen eines Packages
- Die Anzahl von abstrakten und konkreten Komponenten eines Packages (abstrakte und konkrete Klassen, Interfaces)

Aus diesen Angaben werden Kennzahlen ermittelt und Anhaltspunkte zur Interpretation gegeben. Das sind u. a. :

- Abstraktheit (*abstractness*): das Verhältnis der Anzahl von abstrakten Klassen und Interfaces zur Anzahl von konkreten Klassen
- Instabilität (*instability*): das Verhältnis der Anzahl von eingehenden Abhängigkeiten zur Gesamtzahl von Abhängigkeiten, als Maß für die Anfälligkeit des Packages für Änderungen
- Feststellung von zyklischen Abhängigkeiten zwischen Packages

Die Interpretation der gemessenen Werte oder der Kennzahlen und das Finden von Konsequenzen obliegt dem Anwender. Hinweise zur Interpretation finden sich jedoch in [Mar95].

Die ermittelten Werte können z. B. zur Analyse von Modulen verwendet werden. Zum Verständnis: JDepend versteht unter einem Modul eine Menge von Java-Packages. Verwendet wurde für die Analyse ein JDepend-Plugin für Eclipse. Metriken werden dort in einer eigenen Perspektive[66] angezeigt. Dabei kann man auch für mehrere Packages zusammengefaßt Metriken berechnen lassen und auf diese Weise Module untersuchen.

2.3.3 Refactoring

Wie im Abschnitt zu Programmverstehen und -modifizieren beschrieben, verfügt die IDE Eclipse über Fähigkeiten, um zumindest oft benutzte *Refactorings* auszuführen. Sie wird also auch als Werkzeug für diese Aufgabe benutzt. Wichtige unterstützte *Refactorings* sind:

- Umbenennung von Attributen, Methoden, Klassen und Interfaces, Packages
- Extraktion von Methoden, lokalen Variablen
- Kapselung von Attributen
- Verschiebung von Methoden zu Superklassen (*pull up*)

[65] http://www.clarkware.com/software/JDepend.html
[66] Siehe zum Perspektivebegriff die Dokumentation zu Eclipse

Refactoring war ein Thema in dieser Arbeit, nicht das entscheidende Thema. Fehlende Unterstützung für *Refactorings* war deshalb kein Kriterium, eine andere IDE mit vollständigerer Unterstützung zu wählen (z. B. IDEA[67]).

2.3.4 Sonstige

Die bereits erwähnten Werkzeuge wurden in mehr oder weniger großem Umfang eingesetzt. Darüber hinaus wurden weitere Werkzeuge auf ihre Verwendbarkeit untersucht. Im Gegensatz zur ersten Gruppe wurden sie jedoch aus verschiedenen Gründen nicht benutzt. Sie sollen hier jedoch noch angeführt werden.

Macker

Macker[68] ist ein Werkzeug, um die Befolgung von Codierkonventionen oder sogar Designregeln zu prüfen. Untersucht werden „legale“ oder „illegale“ Beziehungen (Assoziationen oder Vererbung; im Macker-Sprachgebrauch Zugriffe genannt) zwischen Klassen. Die Prüfung wird durch Regeln definiert. Regeln beschreiben Muster von Zugriffen. Das tatsächliche Auftreten solcher Zugriffe wird durch Textvergleich zwischen Muster und Package- oder Klassennamen festgestellt. Jede Regel legt darüber hinaus fest, ob der Zugriff erlaubt ist oder nicht.

CodeCrawler

CodeCrawler[69] ist ein Werkzeug zur Visualisierung von Software. Quellcode wird analysiert und in ein sprachunabhängiges, objektorientiertes Metamodell übersetzt. Die so gewonnenen Informationen können auf verschiedene Weise visualisiert werden.

Für Java ist eine Methode dokumentiert, um den Code zu parsen und in das FAMIX-Metamodell zu übersetzen, mit dem CodeCrawler arbeitet. Diese Methode benutzt die IDE SNiFF+ und das Importprogramm sniff2famix. Leider war die angegebene Softwarekonfiguration (SNiFF+ Version 3.2) nicht verfügbar. Mit den verfügbaren Versionen 3.1 und 4.1 von SNiFF+ arbeitete sniff2famix nicht zusammen. So gelang es nicht, den Java-Quellcode mit CodeCrawler zu bearbeiten. Nachfragen per e-Mail bei den Autoren blieben leider ohne Antwort.

SourceNavigator

SourceNavigator[70] ist eine freie IDE für verschiedene Sprachen, darunter auch Java. Sie verfügt über umfangreiche Funktionen zur Navigation in Quellcode. SourceNavigator wurde nicht verwendet, da mit Eclipse eine gleichwertige und besser bekannte IDE bereitstand.

[67] http://www.intellij.com/idea/
[68] http://sourceforge.net/projects/macker
[69] http:://www.iam.unibe.ch/~scg/Archive/Software/CodeCrawler/
[70] http://sourcenav.sourceforge.net

ShrimpView

ShrimpView[71] ist ein Werkzeug zur Visualisierung von Java-Programmcode. Bestimmte Beziehungen im Code können graphisch verdeutlicht werden. Der Nutzer kann quasi über seinem Code schweben, tiefer oder höher fliegen (Zoom in Packages oder Klassen) und sogar Bildfolgen aufnehmen.

ShrimpView wurde nicht verwendet, da das Programm einige Instabilitäten aufwies. So kam es bereits beim Einlesen des TESSI 1.1-Codes unter bestimmten Umständen zu Abstürzen. Auch im Programmverlauf gab es Handlungen, die zu Fehlern führten. Die Entscheidung gegen ShrimpView fiel auch, weil die für diese Arbeit interessanten Zusammenhänge durch die weiter oben genannten Werkzeuge erfaßt werden konnten.

[71] http://shrimp.cs.uvic.ca/shrimp/shrimp_intro.shtml

3. SOFTWAREARCHITEKTUR

Die Einführung in das *Reengineering* aus dem letzten Kapitel hat Wege gezeigt, wie man die Qualität eines bestehenden Systems verbessern kann. Als Ziel der Tätigkeiten wurde die Verbesserung der Wartbarkeit sowie anderer Eigenschaften genannt. Ein Mittel zur Erreichung dieses Ziels ist die Restrukturierung der Systemkomponenten und ihrer Beziehungen, um zu einer bestimmten Architektur für das System zu kommen.

Das ist Grund genug, um sich in diesem Kapitel näher mit dem Begriff „Softwarearchitektur“ zu beschäftigen. Zunächst wird in Abschnitt 3.1 der Begriff bestimmt und sein Umfeld betrachtet. Der Abschnitt 3.2 geht auf das Verhältnis von Architektur und *Refactoring* ein. Abschnitt 3.3 beschäftigt sich dann mit einem Teilgebiet der Softwarearchitektur, das für diese Arbeit von Bedeutung ist.

Der Begriff „Softwarearchitektur“ bezeichnet ein großes und wichtiges Gebiet für die praktische Softwareentwicklung und die Forschung. Es würde den Rahmen dieser Arbeit bei weitem sprengen, auf alle Bereiche einzugehen. Statt dessen wird versucht, Softwarearchitektur einzugrenzen auf Teile, die für diese Arbeit von Bedeutung sind.

3.1 Begriffsbestimmung und Einordnung

Die Architektur eines Softwaresystems beschreibt „... die strukturierte oder hierarchische Anordnung der Systemkomponenten und ihre Beziehungen untereinander.“[1] Unter Systemkomponenten sind abgegrenzte Teile der Software, wie z. B. Funktionen, Prozeduren, abstrakte Datentypen bzw. Klassen usw. zu verstehen. Zu den Beziehungen werden sämtliche statische oder dynamische Verbindungen zwischen Systemkomponenten gerechnet.[2] In [Kro97] werden zusätzlich Struktur und Beziehungen von Daten (und Datenstrukturen) und Funktionalität (prozedurale Komponenten) unterschieden.[3]

Je größer ein Softwaresystem wird, desto sinnvoller und wichtiger wird es, Abstraktionen zu einer stärkeren Strukturierung einzusetzen. Beispiele für solche Abstraktionen sind Schichtenarchitekturen. Die Schichtenarchitektur teilt, wie der Name schon sagt, das Softwaresystem in Schichten ein. Die Zuordnung von Systemkomponenten zu Schichten kann durch verschiedene Kriterien erfolgen, z. B. durch den Abstraktionsgrad. Kennzeichnend für die Schichtenarchitektur sind die Zugriffsregeln für Komponenten in der gleichen Schicht (in der Regel unbeschränkt) und in unterschiedlichen Schichten (Zugriff häufig nur auf Komponenten der nächstniedrigeren Schicht).[4] Für Modularchitekturen ist

[1] [Bal00, S. 699]
[2] [Bal00, S. 696]
[3] [Kro97, S. 113 f]
[4] [Bal00, S. 696 f]

ebenfalls die Steuerung des Zugriffs auf andere Komponenten kennzeichnend.

Eine Architektur für ein Softwaresystem ist ein Ergebnis der Entwurfsphase im Softwareentwicklungsprozeß. In [Bal00, S. 696] wird das Ziel dieser Phase beschrieben als „... für das zu entwerfende Produkt eine Software-Architektur zu erstellen, die die funktionalen und nicht-funktionalen Produktanforderungen sowie allgemeine und produktspezifische Qualitätsanforderungen erfüllt und die Schnittstellen zur Umgebung versorgt.“ Auch im *Reengineering* kann es zum Entwurf einer Architektur kommen: entweder als abstrakte Beschreibung einer bereits existierenden Software oder als Festlegung eines Sollzustands vor der Restrukturierung einer gleichfalls bereits existierenden Software.[5] Vergleicht man das Zitat vom Beginn des Absatzes mit dem *Reengineering*, so fallen Parallelen auf. Das Endprodukt eines *Reengineering*-Prozesses muß ebenfalls alle Produktanforderungen erfüllen. Zu den angesprochenen Qualitätsanforderungen gehören in beiden Prozessen auch die Wartbarkeit und die Änderbarkeit.

Die Qualität einer Softwarearchitektur kann anhand bestimmter Merkmale eingeschätzt werden. Eine Liste solcher Merkmale ist z. B. [Kro97, S. 114] zu entnehmen. Einige Beispiele sind:

- Dekomposition des Gesamtproblems in kleinere Teile und Zuordnung dieser Teile zu Systemkomponenten[6]
- Einhaltung des Geheimnisprinzips (*information hiding*): biete öffentliche Schnittstellen, aber geheime Implementierungen.[7]
- Kapselung von Daten bzw. Gruppierung von Daten und den zu ihrer Bearbeitung benötigten Prozeduren
- Wahl einer optimalen Granularität der Systemkomponenten. Sie sollten nicht zu klein sein, sonst erhöht sich die Komplexität der Abhängigkeiten. Sie sollten allerdings auch nicht zu groß sein, damit sie beherrschbar bleiben.[8]

In [WBM94, S. 10 ff] werden noch zusätzliche Punkte als „Axiome des Designs“ genannt:

- Getrennte Bearbeitung von Systemaspekten („*Axiom of the separation of concerns*“)
- Komponenten sollten nicht von Annahmen über ihre Umgebung abhängen, die über die spezifizierte Schnittstelle hinausgehen. Sie können daher ohne Veränderung in jedem äquivalenten Kontext eingesetzt werden. („*Axiom of translation*“)
- Komponenten sollten ihrerseits austauschbar sein („*Axiom of transformation*“)

[5] Das Aufdecken einer unbekannten Architektur liegt in der Verantwortung des *Reverse Engineering*. Die Veränderung der aufgedeckten Architektur kann nach Definition des Begriffes ebenfalls als Restrukturierung bezeichnet werden (siehe Definition auf Seite 12).

[6] [WBM94] nennt als Maxime das seit langem bewährte *Divide et impera*.

[7] Siehe auch [WBM94, S. 17].

[8] [Kro97, S. 114] nennt als Empfehlung einen Aufwand von 10 Personentagen zum Implementieren und Testen einer Komponente.

Eine Softwarearchitektur beschreibt jedoch nicht nur die Struktur und die Beziehungen von Komponenten im System. Der Entwurf der Architektur ist auch von verschiedenen Umgebungsbedingungen des Systems abhängig, z. B. die zu benutzende Plattform, Technologien, Systeme, mit denen kommuniziert werden muß usw. Diese Rahmenbedingungen spiegeln sich im Resultat natürlich wider.

Das Finden von Lösungen für Architekturprobleme ist eine schwierige Aufgabe. Es muß nicht nur das Problem an sich gelöst werden, sondern auch eine gute Lösung (im Sinne von Softwarequalität bzw. im Sinne der angestrebten Eigenschaften von Software) gefunden werden. Schlechte Lösungen bedeuten einen erhöhten Korrekturaufwand. Da liegt der Gedanke nahe, gefundene und bewährte Lösungen wiederzuverwenden. Ein Mittel zur Wiederverwendung von „guten“ Designs sind die Entwurfsmuster (*Design Patterns*).

Die Idee, gefundene Entwurfslösungen als Muster zu beschreiben und später in anderer Umgebung wiederzuverwenden, stammt nicht aus der Softwaretechnik, sondern aus der Architektur.[9] Zu Bedeutung für Software kam sie mit Veröffentlichungen und Büchern wie [GoF96] und [Bus96]. Angewendet wurden Entwurfsmuster aber wahrscheinlich schon viel früher, und zwar intuitiv durch erfahrene Fachleute in vielen Gebieten.

Entwurfsmuster gibt es inzwischen für viele Probleme der Softwarearchitektur. Es gibt Muster für Strukturen von ganzen Systemen. In [Bus96] werden Muster beschrieben, die z. B. die Struktur ganzer Betriebssysteme beschreiben (Schichtenarchitektur, *Microkernel*). Daneben gibt es aber auch Muster für Detaillösungen. *Design Patterns* beschreiben statische Strukturen (*Architectural Patterns* in [Bus96], *Structural Patterns* in [GoF96]) sowie dynamisches Verhalten (*Behavioral Patterns*).

Die verantwortungsvolle Anwendung von Mustern für die Architektur ist jedoch ebenfalls von gesammelter Erfahrung abhängig. Im negativsten Fall kann sich ein angewendetes Muster im nachhinein als schädlich erweisen. Die Korrektur der Architektur hat dann die gleichen Folgen wie ein „normaler“ Designfehler.

3.2 Softwarearchitektur und Refactoring

Wie angedeutet wurde, geben *Design Patterns* dem Entwerfer ein Mittel in die Hand, von bewährten Lösungen erfahrener Fachleute zu profitieren. Es gibt auch Ansätze für den umgekehrten Weg: die Erkennung häufiger Entwurfsfehler und dazugehörige Lösungen.

Durch die Softwarearchitektur wird im Entwurf eine Struktur vorgegeben, die der Programmierer mit der Implementierung zu füllen hat. Wie im Abschnitt 2.2.3 beschrieben wurde, haben Restrukturierung und *Refactoring* einen völlig anderen Ausgangspunkt: sie gehen vom vorhandenen Code aus. Es wird versucht, den Code zu verstehen und eine bessere Struktur zu finden. Schließlich wird der Code solange transformiert, bis der gewünschte Zustand erreicht ist. *Refactorings*, wie sie in [Fow99] beschrieben sind, oder die *Reengineering Patterns* aus [DDN03] liefern sowohl Hinweise für das Finden von Designfehlern, als auch Lösungswege zur Korrektur.

[9] Ein Architekt namens Christopher Alexander schrieb Ende der 1970er Jahre Bücher über Muster und deren Anwendung in der Architektur von Gebäuden, Wohngebieten und Städten.

3.3 Softwarearchitektur im Projekt TESSI

Im Projekt TESSI wird eine objektorientierte Anwendung entwickelt, die über eine graphische Benutzerschnittstelle (GUI) verfügt. Bereits diese Information schränkt die Grobauswahl der Architektur ein, da die Kategorie der Anwendung den Architekturentwurf determiniert.[10] Es gibt verschiedene solcher Kategorien: *Desktop*-Anwendungen, *Client/Server*-Anwendungen, *Web*-Anwendungen usw. TESSI ist demnach eine *Desktop*-Anwendung.

Eine generelle Struktur für solche Anwendungen gibt die 3-Schichten-Architektur vor, bestehend aus Fachkonzept-Schicht, GUI-Schicht und Datenhaltungsschicht. Unter dem Namen *n-tier architecture* ist diese Architektur auch für *Web*-Anwendungen bekannt.[11] Kriterium für die Schichtung ist die Zugehörigkeit der Verantwortlichkeiten einer Komponente zu den drei Aufgabenbereichen. Die Komponenten einer Schicht beanspruchen nur Dienstleistungen einer tieferen Schicht.[12]

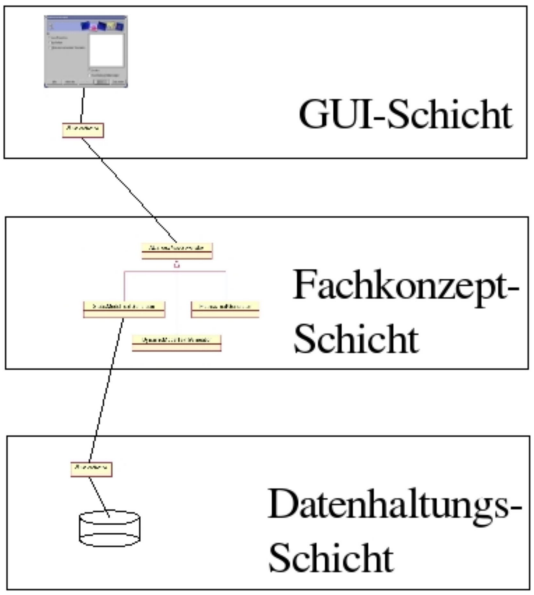

Abb. 3.1: 3-Schichtenarchitektur für *Desktop*-Anwendungen

Abbildung 3.1 verdeutlicht die Architektur. Zu sehen sind die drei übereinander angeordneten Schichten. Jeder Schicht sind Klassen zugeordnet. Zwischen benachbarten Schichten gibt es Kommunikation. Die oberste Schicht entspricht dem, was der Nutzer vom System „zu sehen“ bekommt. Die unterste Schicht realisiert die Kopplung zu einem System, welches die persistente Speicherung der Daten übernimmt (z. B. einer Datenbank).

[10] [Bal00, S. 987].
[11] Meist als *3-tier architecture* angewendet (*client-tier* oder *presentation-tier*, *application-tier* und *data-storage-tier*). Es können aber auch noch mehr Schichten (oder *tiers*) vorhanden sein.
[12] [Bal00, S. 697]

Mit diesem Modell wurde zunächst nur eine grobe Einstufung vorgenommen, die weiter verfeinert werden muß. Eine Beschreibung des Vorgehens für objektorientierte Applikationen mit einer GUI ist in [Bal00, S. 992 ff] enthalten. Demnach wird zunächst das Fachkonzept entworfen bzw. verfeinert. Danach folgen die beiden anderen Schichten. Beim verfeinerten Entwurf der Schichten, d. h. der den Schichten zugerechneten Klassen und Assoziationen, sind weitere Entscheidungen zur Architektur zu treffen.

Klassen des Fachkonzeptes werden z. B. durch Klassen der GUI-Schicht dargestellt bzw. dem Nutzer zugänglich gemacht. Dies kann z. B. durch ein Dialogfenster geschehen. Für diesen Fall ergibt sich eine Konstellation, die durch das Entwurfsmuster „*Model-View-Controller*“ beschrieben wird.[13] An der Erwähnung des Musters ist auch gleich zu erkennen, daß für solche Fragen bereits Lösungen existieren. Der Entwerfer sollte sie kennen und bei Bedarf anwenden.

Auch in einigen Programmiersprachen, bzw. in den zur Sprache gehörenden (Klassen-)Bibliotheken, findet man *Design Pattern*. Für TESSI wurde als Implementationssprache Java festgelegt. Viele Klassen und Klassenkonstellationen, die in der Standardklassenbibliothek von Java definiert sind, wurden nach *Design Patterns* entworfen. Beispiele sind Iteratoren oder das *Listener*-Konzept in der Swing-Bibliothek. Dieser Umstand kann sich positiv auswirken, wenn es möglich ist, die in Java geleistete Entwurfsarbeit für eigene Entwürfe zu nutzen.

[13] Das MVC-Entwurfsmuster ist z. B. in [Bus96, S. 125 ff] beschrieben.

4. VORBETRACHTUNGEN

4.1 Was bisher geschah

Textual Assistant heißt ein Forschungsprojekt an der Professur für Informationssysteme und Softwaretechnik der TU Chemnitz. Gegenstand der Forschung sind Möglichkeiten der Unterstützung des Analytikers bei der Analyse der Anforderungen in einem Softwareprojekt. In diesem Forschungsprojekt wird das Programm TESSI entwickelt.

Basis für die vorliegende Arbeit sind verschiedene studentische und andere Arbeiten an TESSI. Eine erste Version der Software wurde im Rahmen einer Diplomarbeit von Mathias Strauß entwickelt (siehe [Str96]). Die Grundlage für die Verbindung zu Rational Rose® bildete die Studienarbeit von Lars Gemeinhardt (siehe [Gem00]). Von Sven Leidenfrost (siehe [Lei00]) stammt der weitaus größte Teil der Version 1.1 von TESSI. Diese Version bildet den Ausgangspunkt für die zu entwickelnde Version 2.0, da sie den zu erreichenden Funktionsumfang definiert.

Die Funktionsfähigkeit der TESSI zugrundeliegenden Idee und von TESSI 1.1 selbst wurde von Lars Rosenhainer in einem Testprojekt in Zusammenarbeit mit der Firma RAWEMA nachgewiesen. TESSI 1.1 wurde ebenfalls für die Lehre eingesetzt.

Für Erweiterungen und Veränderungen an TESSI gab und gibt es eine Reihe von Ideen und Plänen. Diese haben einerseits zum Ziel, den Modellierungsumfang und die Interoperabilität von TESSI zu verbessern. Andererseits soll die Unterstützung, die TESSI einem Analytiker für die Arbeit bietet, durch Automatisierung von Analyseaufgaben und Bereitstellung weiterer Hilfsmittel verbreitert werden.

Speziell auf die erste Gruppe von Erweiterungen (Modellierungsumfang, Interoperabilität) zielte das Vorhaben, die Datenhaltung in TESSI 1.1 durch ein Repository zu ersetzen, welches sich eng an der UML[1] orientieren sollte. Zusätzlich zum Repository sollten die Modelldaten per XMI[2] im- und exportiert werden können. Der Vorbereitung dieses Vorhabens dienten ein Hauptseminar (siehe [Tos01]) und eine Studienarbeit (siehe [Tos02]). Im Rahmen der Studienarbeit wurde der Prototyp eines Repositorys entworfen und implementiert.

Leider war die Fertigstellung nicht möglich, da der Umfang der Arbeit die vorhandenen Ressourcen bei weitem überstiegen hätte. Auch der zu erwartende Aufwand für die Softwarepflege wurde als zu hoch eingeschätzt. Die Notwendigkeit, die angestrebten Veränderungen durchzuführen, blieb jedoch bestehen.

[1] Unified Modelling Language
[2] XML Metadata Interchange

4.2 Geplante Veränderungen

Vor dem *Reengineering* eines Systems kann es wichtig sein, zu wissen, welche Veränderungen kurz- oder mittelfristig anstehen. Entscheidungen zur Umgestaltung können sich als falsch herausstellen, wenn dieser Aspekt vernachlässigt wird. Aus diesem Grund lohnt es sich, die kommenden Veränderungen im Projekt TESSI zu betrachten.

4.2.1 Metamodelle, Modellierungsumfang, Datenaustausch

TESSI 1.1 liegt ein Metamodell zugrunde, das sich an einer (kleinen) Teilmenge der UML orientiert.[3] Trotz aller Ähnlichkeit sind die Modellkonstrukte aus der Schnittmenge nicht gleich. Manche unterschieden sich sogar erheblich voneinander. Ein ausführlicher Vergleich der Konstrukte wurde in [Tos02, Kapitel 3] angestellt. Das Metamodell von TESSI 1.1 wird durch die 17 Klassen im Package `Tessi.Datatypes` implementiert, Dokumentation zum Metamodell befindet sich in [Gem00].

Die Speicherung der Modelldaten erfolgt auf Basis einer Abbildung des Metamodells auf XML.[4] Das Format kann in Rational Rose® importiert und exportiert werden, allerdings nur durch zwei Skripte für Rational Rose®, die von Lars Gemeinhardt stammen. Aliasnamen, die man den Modelldaten zurechnen könnte, werden beim Datenaustausch mit Rational Rose® nicht erfaßt. Ein Datenaustausch mit anderen Werkzeugen ist ohne Konverter nicht möglich. Momentan existieren solche Konverter nicht.

Es war natürlich klar, daß Erweiterungen des Metamodells von TESSI 1.1 auch Erweiterungen der Klassenbibliothek, die das Metamodell implementiert, des Datenformates, des Lade- und Schreibemodules für Modelldaten und der Import/Exportskripte für Rational Rose® nach sich ziehen würden. Modifikationen waren auch für alle Bereiche in TESSI 1.1 zu erwarten, in denen Modelldaten oder deren Eigenschaften angezeigt oder modifiziert werden.

Wollte man dagegen zur UML als Metamodell und zu einem UML-sprechenden Repository zur Datenverwaltung übergehen, waren zwar noch etwas umfangreichere, im Kern aber ähnliche Umbauten vonnöten. Die beiden schon erwähnten Arbeiten [Tos01] und [Tos02] sollten genau dies erreichen, konnten aber wegen des zu erwartenden Aufwandes nicht abgeschlossen werden.

Während die vorbereitenden Arbeiten erstellt wurden, zeichnete sich eine Alternative ab: JMI[5]. Mit der Spezifikation von JMI[6] und den Referenzimplementierungen ergab sich die Möglichkeit, ein Repository „einzukaufen“, statt selbst zu entwickeln.[7]

JMI definiert eine Umsetzung von Metamodellen zu Java-Interfaces sowie Semantik zu deren Implementierung. JMI stützt sich auf MOF[8] als Metame-

[3] Genauer gesagt orientiert es sich am Metamodell von Rational Rose®, welches wiederum sehr ähnlich zur UML ist.

[4] Dokumentiert in [Gem00]

[5] Java Metadata Interface

[6] Siehe Homepage des Standardisierungsprozesses: `http://jcp.org/en/jsr/detail?id=40` sowie die Produkthomepage: `http://java.sun.com/products/jmi/`

[7] Es gab bereits vorher Pläne, vorhandene, CORBA-basierte Repositorys zu nutzen. So wurde die Software dMOF des DSTC (Cooperative Research Centre for Enterprise Distributed Systems Technology) evaluiert. Die Nutzung war aber aus finanziellen Gründen ausgeschlossen. MOF-Homepage des DSTC: `http://www.dstc.edu.au/Research/Projects/MOF`

[8] Metaobject Facility

tamodell und auf XMI als standardisiertes Austauschformat. JMI nimmt hier die Rolle ein, die in der MOF-Spezifikation CORBA[9] zugedacht wird. JMI ist speziell auf die Nutzung eines Repositorys aus Java heraus konzipiert. Auf eine *Middleware*, wie CORBA, bezieht sich die Spezifikation dagegen nicht.[10]

Die Spezifikation von JMI, die bei Anfertigung dieser Arbeit benutzt wurde, ist als *Final Specification* (Version 1.0 vom 07. Juni 2002) gekennzeichnet. Dazu existieren bereits mindestens zwei Implementierungen.[11] Beide wurden in Vorbereitung dieser Arbeit untersucht. Ausgewählt wurde schließlich das *Metadata Repository* des Netbeans-Projektes. Mit dem Repository von Unisys war es zuvor nicht gelungen, eine Instanz eines Repositorys zur UML 1.3 anzulegen. Die Entscheidung für die UML Version 1.3 war zuvor gefällt worden, da es zur Zeit nur für diese Version Austauschmöglichkeiten mit Rational Rose® über XMI gibt.

Am Ende der Überlegungen stand schließlich der Entschluß, gleichzeitig mit dem Austausch der Datenhaltung eine Restrukturierung von TESSI 1.1 in Angriff zu nehmen, um es für kommende Erweiterungen fit zu machen.

4.2.2 Nutzerschnittstelle

Bei der Arbeit mit TESSI 1.1 ergab sich der Wunsch nach einer Verbesserung der Nutzerschnittstelle. Einige Funktionen waren z. B. schlecht erreichbar. Der Anteil der GUI am gesamten Code ist sehr hoch. Da bei einer Veränderung des Metamodells sowieso umfangreiche Modifikationen zu erwarten waren, sollten die Veränderungen der Nutzerschnittstelle gleich mit durchgeführt werden.

4.2.3 Wartung

Erfahrungen in der Wartung ergaben, daß TESSI 1.1 an vielen Stellen sehr komplex und schwer zu verstehen war. Darin lag der Grund für Schwierigkeiten beim Beseitigen von Fehlern. Eine nötige Anpassung an eine neuere Version des verwendeten Thesaurus WordNet führte zu einer ersten Auslagerung von Komponenten.

4.2.4 Dokumentation

Die Diplomarbeit von Sven Leidenfrost (siehe [Lei00]), in der TESSI 1.1 hauptsächlich entstand, enthält umfangreiche Informationen zu den verwendeten Konzepten und dem Metamodell sowie eine ausführliche Benutzerdokumentation. Die Dokumentation zum Entwurf ist nicht so umfangreich. Um diesem Mangel abzuhelfen, wurde TESSI 1.1 *reengineered*. Es entstand als Ergebnis ein entsprechendes Rational Rose®-Projekt. Die vorhandene Quellcodedokumentation war teilweise nicht Javadoc-konform und wurde mit Skripten nachbearbeitet.

Damit bei folgenden Versionen von TESSI keine unnötigen Schwierigkeiten beim Verständnis mehr auftreten würden, war die vollständige Dokumentation von Architektur und Quellcode eine Hauptforderung für zukünftige Projekte.

[9] Common Object Request Broker Architecture

[10] Zu MOF, UML, XMI und den Zusammenhängen zwischen diesen OMG-Standards siehe [Tos01, Kapitel 1 und 2].

[11] Links zur Referenzimplementierung von Unisys und der Open-Source-Implementierung MDR im Netbeans-Projekt finden sich auf der JMI-Produkthomepage.

4.3 Untersuchung der Architektur im aktuellen TESSI

Bei der Analyse des IST-Standes von TESSI Version 1.1 und der Suche nach Verbesserungsmöglichkeiten kamen die in Abschnitt 2.3 vorgestellten Werkzeuge zum Einsatz. Ausgewählte Ergebnisse werden hier erläutert.

Bei den Erläuterungen wird vorausgesetzt, daß der Leser mit der Funktion von TESSI 1.1 vertraut ist. Diese ist in [Lei00] ausführlich beschrieben.

4.3.1 Reverse Engineering

Teile von TESSI 1.1 wurden bereits vor dieser Arbeit im Rahmen der Wartung einem *Reengineering* unterzogen. Das Projekt wurde von Lars Rosenhainer durchgeführt. Da wesentliche Informationen über TESSI 1.1 vorher undokumentiert waren, wurde das Programm *reverse engineered*. Das Ergebnis liegt als Projekt für Rational Rose® vor und wurde für diese Arbeit mit genutzt.

4.3.2 Modulaufbau

Ein Problem war bereits vorab bekannt: TESSI 1.1 ist nicht (bzw. fast nicht) modular aufgebaut. Was aber ist eigentlich Modularisierung, was ist ein Modul? In [Mue97, S. 14] wird der Begriff Modularisierung wie folgt definiert:

> **Modularisierung (Modularization)**: Unter Modularisierung versteht man die Partitionierung eines Monolithen in seine funktional zusammenhängenden Module. Durch die geringe Größe der entstehenden Module sind diese besser zu überblicken, zu verstehen und zu warten.

In [GJ97, S. 242] werden Module als Mechanismus zur Abstraktion und Dekomposition beschrieben. Abstrahiert wird von der konkreten Umsetzung von Diensten, die für den Nutzer der Dienste verborgen bleibt. Ein bereitgestellter (exportierter) Dienst eines Moduls wird mit Mitteln der Programmiersprache beschrieben, also teilweise spezifiziert.

Viele Programmiersprachen sehen keine Sprachkonstrukte explizit für die Deklaration und Definition von Modulen vor. Es gibt die Meinung, daß in objektorientierten Sprachen Klassen als Module anzusehen sind.[12] Anhand der Konzepte von Programmiersprachen wie Ada oder Modula-2 kann man den Modulbegriff auch erweitert auffassen, etwa als Subsystem.

Für Module gelten u. a. folgende Anforderungen:[13]

1. **Kapselung** von Daten und Diensten; ermöglicht die Gruppierung von Komponenten, die zur Erbringung eines Services zusammenarbeiten; ermöglicht weiterhin das Verbergen irrelevanter Details.[14]

2. Definition einer **Schnittstelle** des Moduls, welche die exportierten Dienste beschreibt.

[12] [GJ97, Kapitel 5.2 und 5.3 ab S. 241]
[13] Es gibt weitere Eigenschaften. Ein Beispiel ist die Möglichkeit, Module getrennt voneinander zu entwickeln und zu kompilieren. Ein Modul, das Dienste importiert, benötigt nur die Schnittstelle des exportierenden Moduls.
[14] [GJ97, S. 243]

3. Trennung von **Schnittstelle** und **Implementierung**; Klienten für Modulservices können unabhängig von der Implementierung der Dienste entwickelt werden. Änderungen in der Implementierung haben so nur modullokalen Einfluß, solange die Schnittstelle eingehalten wird.

Klassen, sowie Klassen in Verbindung mit der Paket-weiten Sichtbarkeit ermöglichen in Java die Realisierung beider genannter Modulbegriffe.[15] Werkzeuge wie JDepend basieren auf der Annahme, daß ein Modul aus dem Inhalt eines (oder mehrerer) Java-Packages bestehen kann.[16] Die Schnittstelle des Moduls ist dabei die Summe der Schnittstellen öffentlicher Klassen oder Interfaces in den zum Modul gehörenden Packages. Im Weiteren wird diese Auffassung des Modulbegriffes benutzt.

Wie oben schon angedeutet, ist TESSI 1.1 nicht explizit modular aufgebaut. Einerseits hat es der Autor nicht so geplant. Andererseits existiert auch kein entsprechendes Sprachkonstrukt[17] dafür in Java und es gibt auch keine anderen Dokumente, die eine Modulstruktur beschreiben. Indizien sprechen aber dafür, daß zumindest in zwei Fällen Komponenten der Software als Module „gedacht waren“.

TESSI 1.1 stammt zum größten Teil von Sven Leidenfrost. Es gibt jedoch zwei Bereiche, die von anderen Autoren stammen. Diese Ausnahmen sind:

1. Alle Klassen, die das Speichern und Laden der Modelldaten in XML-Dateien realisieren (Packages `Tessi.XML` und Unterpackages).[18] Autor ist Lars Gemeinhardt.

2. Das Package `de.tuc.isst.tessi` und seine Unterpackages sind das Ergebnis einer Restrukturierung. Die meisten enthaltenen Klassen realisieren die Ansteuerung des Thesaurus WordNet. Die Restrukturierung wurde von Lars Rosenhainer durchgeführt.

Aufgrund der getrennten Entwicklung und der in sich relativ abgeschlossenen Funktion kommen diese Komponenten am ehesten als Kandidaten für Module in TESSI 1.1 in Frage. Wegen dieser Annahme wurden die Kopplungen zwischen Klassen aus verschiedenen Packages mit JDepend analysiert. Gruppiert man die Pakete mit den meisten Kopplungen und faßt diese als Module auf, erhält man das Modulmodell in Abbildung 4.1. Welche Java-Packages den Modulen zugerechneten werden, kann den Kommentaren entnommen werden.

Die geäußerte Vermutung wird durch die Ergebnisse der Untersuchung erhärtet und soll im Weiteren als bestätigt angesehen werden.

In [Tos02, Kapitel 3] wurde die Datenhaltung in TESSI 1.1 untersucht. Ein solches Modul ist, wie bisher deutlich geworden sein sollte, nicht als solches definiert. Aufgrund der Namensgebung kann man aber annehmen, daß die drei

[15] [GJ97, S. 274 f]

[16] Tatsächlich werden in der Dokumentation zu JDepend nirgends die Begriffe *module* und *package* explizit zusammengebracht. Es findet sich statt dessen der folgende Satz: *Packages that are cohesive and independent can be released as autonomous modules with their own release schedules and version numbers.* Dieser Satz legt nahe, daß hier der Modulbegriff auf Gruppen von Java-Packages ausgedehnt wird.

[17] Man könnte `package` als ein solches Sprachkonstrukt ansehen. Die Konstrukte `package` (und `package body`) von Ada sind im Vergleich dazu aber eindeutiger und restriktiver. Die dort enthaltenen Informationen muß man sich in Java von potentiell vielen Stellen zusammensuchen.

[18] TESSI 1.1 speichert Modelldaten in einem eigenen, auf XML basierenden Format.

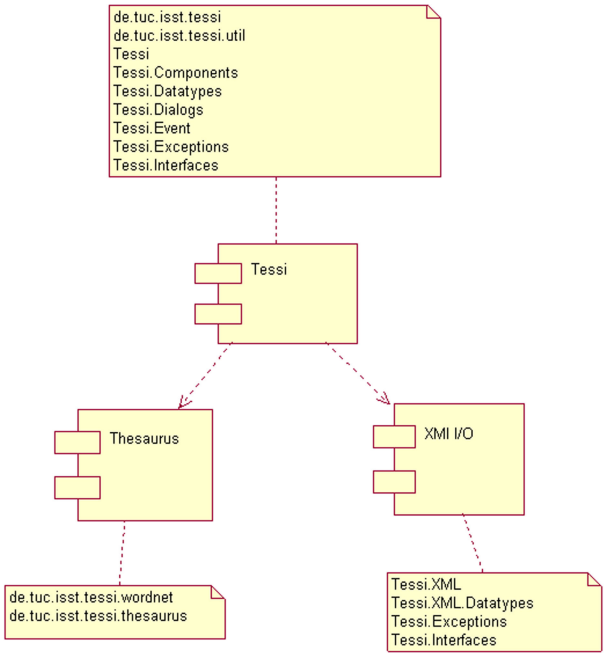

Abb. 4.1: Erster Ansatz eines Modulmodells in TESSI 1.1

Klassen `Tessi.DRModel`, `Tessi.DRText` und `Tessi.DRTree` einem gedachten Modul „Datenhaltung“ zugeordnet werden können (alle drei haben das Präfix `DR`, vielleicht für *data repository*). Es wird in [Tos02] ebenfalls darauf hingewiesen, daß die Klasse `Tessi.DRTree` lediglich eine Abstraktion der Modelldaten als Baum definiert. Sie „hält“ also keine zusätzlichen Daten und soll daher im Gegensatz zur genannten Quelle nicht mehr diesem gedachten Modul zugerechnet werden.

Die Klasse `Tessi.DRText` ist relativ groß (die viertgrößte Klasse in TESSI 1.1 überhaupt) und enthält in großer Zahl Methoden zur Generierung verschiedener Textarten aus vorhandenen Modellkonstrukten. Eine weitere Gruppe von Methoden realisiert die Hervorhebung der Namen von Modellelementen im Spezifikationstext. Beide Gruppen tragen nicht zur Speicherung von Daten bei und werden daher der Datenhaltung ebenfalls nicht zugerechnet. Schließlich gibt es `get`-Methoden für den Zugriff auf die Texte selbst.

`Tessi.DRModel` besitzt viele Methoden zum Erzeugen, Zerstören und Bearbeiten der Modellelemente, die nicht Teil anderer Modellelemente sind, sowie eine Anzahl von Suchfunktionen nach Eigenschaften einiger Modellkonstrukte. Der Wert der LCOM*-Metrik (0,996, ermittelt mit dem Eclipse-Plugin net.sourceforge.metrics) für diese Klasse läßt vermuten, daß eine Aufteilung auf kleinere Klassen eine Verbesserung der Struktur mit sich bringen könnte.[19] Auf-

[19] Allgemein weisen die Klassen in TESSI 1.1 einen LCOM* Wert nahe 1 auf. Laut Dokumentation kann dies ein Indiz für die notwendige Aufteilung einer Klasse sein. LCOM* wird in dem Buch „*Object-Oriented Metrics, Measures of Complexity*“ von Brian Henderson-Sellers, Prentice Hall, 1996 detailliert beschrieben, das dem Autor dieser Arbeit jedoch nicht vorlag. Quellen zur verwandten LCOM-Metrik äußern sich aber ähnlich (z. B. [FAM99, S. 278 ff]).

grund der Art der Methoden — es handelt sich um eine Art *convenience functions*, die die Navigation in den Modellstrukturen erleichtern sollen — scheint hier aber die Deklaration einer separaten Adapterklasse besser.

Das Schreiben und Lesen der Modelldaten und Texte wird von einigen Methoden in der Klasse `Tessi.MainFrame` erledigt, die direkt durch eine Eventroutine gesteuert sind (`Tessi.Mainframe.actionPerformed`). Hier scheint eine Auslagerung in eine Controllerklasse angebracht, die dem Modul Datenhaltung zugerechnet werden sollte.

Neben den genannten wurden noch andere Veränderungen identifiziert, deren Durchführung zur Separierung eines weiteren Moduls notwendig sein würde. Sie sollen nicht im Einzelnen erläutert werden.

Der Grund dafür liegt in einer Anforderung dieser Arbeit, die im Abschnitt 4.1 schon angedeutet wurde: die Anpassung der Modellierfähigkeiten von TESSI an die UML. Ebenfalls beschrieben wurde, daß ein entsprechendes Projekt wegen des hohen Aufwandes nicht zu Ende geführt werden konnte. Es wurde im Weiteren entschieden, die Datenhaltung komplett durch eine andere Software zu ersetzen. Die gewonnenen Analyseergebnisse waren aber trotzdem wichtig für die Identifizierung des wiederzuverwendeten Codes.

Der aus der Klasse `Tessi.DRText` stammende Code zur Generierung von Texten ist relativ umfangreich und bildet intuitiv eine funktionelle Einheit. Es wurde daher entschieden, ein Modul „Textgenerierung“ zu definieren.

TESSI 1.1 bietet die Möglichkeit, Metriken über erzeugte Modelldaten zu erstellen. Konkret besteht diese Funktion darin, die Anzahl der einzelnen Modellkonstrukte in einen generierten Text auszugeben. Die Funktion wird durch die Klasse `Tessi.Metrics` realisiert. Die Idee liegt nahe, auch den Metrikentext in das Modul „Textgenerierung“ zu integrieren. Zwar besteht die Möglichkeit einer Erweiterung auf andere Metriken und Ausgabeformate. In diesem Fall ist es aber einfach, das Modul später aufzuteilen.

Die Erzeugung, Zerstörung und Bearbeitung von Modelldaten wird in TESSI 1.1 von einer Reihe von Klassen realisiert, deren Verantwortlichkeit folgenden vier Gruppen zugeordnet werden kann:

1. Hilfsklassen, die GUI-Elemente mit gegenüber der Java-Swing-Bibliothek erweiterten Fähigkeiten definieren (einige mit Namenspräfix `TD`, weitere im Package `Tessi.Components`
2. Hilfsdatentypen zur Unterstützung der Anzeige von Modelldaten (Namenspräfix `THD`)
3. Graphische Komponenten (Panels, Dialoge, Toolbars,...)
4. Separate Dialoge (die sogenannten ExpansionDialoge, zu finden im Package `Tessi.Dialogs`)

Der Code zur Manipulation der Modelldaten befindet sich direkt in der Präsentationslogik der GUI-Klassen. Er ist überdies auf viele Klassen verteilt. Abbildung 4.3 auf Seite 45 zeigt beispielhaft die Verteilung von Code zur Erzeugung und Zerstörung von Instanzen des Modellkonstruktes Klasse.

Es ist klar, daß die Manipulation von Modelldaten ein wichtiger Aspekt von TESSI ist. Es wurde entschieden, auch diese Funktionen in einem Modul zusammenzufassen. Die Wiederverwendbarkeit des betreffenden Codes aus TESSI 1.1 war allerdings als gering einzustufen. Die Manipulation der Modelldaten ist stark abhängig von den Klassen, die die Modellkonstrukte implementieren. Da dieser Teil von TESSI 1.1 ersetzt werden würde, war der bestehende Code praktisch nicht zu verwenden. Ähnliches gilt für die Hilfsdatentypen. Die graphischen Komponenten zur Eingabe von Modelldaten und deren Eigenschaften richten sich natürlich ebenfalls an den Modelldaten aus. Für TESSI 2.0 sollte außerdem die Benutzerschnittstelle verändert werden. Für diese Änderungen waren die vorhandenen Komponenten in der Regel nicht flexibel genug.

Für die Expansiondialoge bestand die Chance der Wiederverwendung mit geringen Modifikationen. Diese bestanden darin, keine direkte Manipulation des Modells mehr darüber durchzuführen.

Es wurde also entschieden, ein Modul „Modellmanipulation“ vorzusehen, welches größtenteils neu zu implementieren war.

Übrig bleiben im Wesentlichen drei Funktionen: das Editieren eines Spezifikationstextes, die Darstellung der vorhandenen Modelldaten als Baum und das Anzeigen von Hilfe, Legende und Kontexthilfe.

Die Lösung zur Textbearbeitung aus TESSI 1.1 stützte sich stark auf vorhandene Java-Klassen der Swing-Bibliothek. Dies sollte beibehalten werden. Sehr abhängig vom verwendeten Editor war die Hervorhebung von Namen von Modellelementen im Text. Beide Funktionen wurden einem Modul „Spezifikationstext“ zugewiesen.

Die Baumdarstellung der Modelldaten stützt sich ebenfalls stark auf Komponenten von Java. Auch hier sollte die Lösung beibehalten werden. Nach Anpassung an das neue Datenmodell scheint dieser Bereich am wenigsten von zukünftigen Änderungen betroffen zu sein. Diese Gründe führten zur Entscheidung für ein Modul „Modellübersicht“.

Die Hilfe sollte insgesamt aus TESSI 2.0 ausgelagert werden. Die Texte sollen weiterhin aus HTML-Dateien bestehen und durch einen externen Webbrowser dargestellt werden. Das Modul „Hilfe“ unterstützt die Anzeige der Texte, indem es auf Anforderung anderer Module die notwendigen Aktionen ausführt.

Mit den Überlegungen aus diesem Abschnitt wurde ein Modulmodell entwickelt, welches die in TESSI 1.1 realisierten Funktionen logisch zuordnet (siehe Abbildung 4.2).

4.3.3 Separation of Concerns

Zum Bereich dieser Arbeit gehört auch eine Untersuchung, welchen Erfolg die Zerlegung des Problems in Teilprobleme hatte und welche anderen Belange unter dieser Zerlegung zu leiden hatten. Ziel der Zerlegung ist es eigentlich, die Teilprobleme getrennt zu lösen (*separation of concerns*). Mit den Modularisierungstechniken, die gängige objektorientierte Sprachen heute bieten, ist jedoch eine Zerlegung, die allen Teilproblemen gerecht wird, nicht möglich. Fortgeschrittene Techniken wie die Aspektorientierung versuchen dieses Problem zu lösen.

Mit Blick auf die Evolution von TESSI soll der IST-Stand festgestellt werden. Anhand von Beispielen wird so ein Vergleich mit TESSI 2.0 möglich.

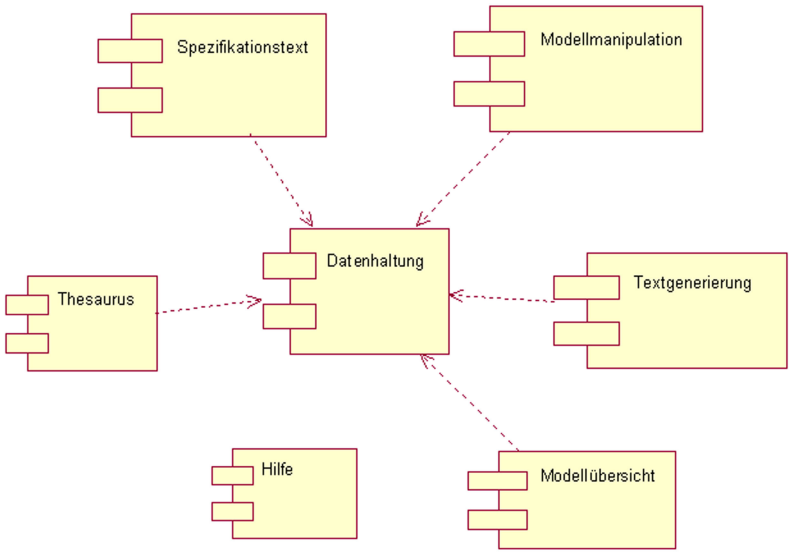

Abb. 4.2: Modulmodell zu den in TESSI 1.1 aufgefundenen Funktionen

Für die Zerlegung von *concerns* führt die Aspektorientierte Programmierung das Konzept der Aspekte ein. Aspekte dienen der separaten Behandlung von *concerns*, die in herkömmlichen Programmiersprachen nicht separierbar gewesen wären (sogenannte *hidden concerns*). *Hidden concerns* sind daran zu erkennen, daß der implementierende Code über viele Stellen verteilt ist und/oder mit Code zu anderen *concerns* verwoben ist.[20]

Für die Suche nach *concerns* in Quellcode gibt es den Begriff *aspect mining* und ein entsprechendes Werkzeug, das *Aspect Mining Tool* (AMT). Das Werkzeug wurde bereits in Abschnitt 2.3.2 auf Seite 23 beschrieben.

Eine gewisse Vorstellung davon, welche *concerns* im Projekt überhaupt vorhanden sein könnten, ist sehr hilfreich oder sogar Voraussetzung für die Arbeit mit AMT. Für diese Arbeit soll zunächst die Existenz einiger *concerns* behauptet werden. Mittels AMT sollen diese dann im Quellcode von TESSI 1.1 gesucht und deren Zustand eingeschätzt werden. In einem späteren Abschnitt wird diese Analyse mit dem Quellcode von TESSI 2.0 wiederholt.

Gestützt auf Wissen über TESSI und Erfahrungen bei der Analyse des Quellcodes von TESSI 1.1 wurden die folgend genannten *concerns* zur genaueren Untersuchung ausgewählt. Zu jedem Punkt ist eine kurze Begründung und ein Konzept zur Erkennung des dazugehörigen Codes angeführt. Abschließend werden die Analyse und ihr Ergebnis beschrieben.

Zum Vorgehen ist anzumerken, daß es sich hierbei nicht um echtes *Mining* handelt. Der Umfang des Quellcodes von TESSI 1.1 ist noch überschaubar, seine Funktion bekannt. Es wird hier eher nach Beweisen für begründete Verdächte gesucht.

Bezogen auf die Abbildungen sei darauf hingewiesen, daß immer nur ein Teil der Fundstellen zu sehen ist. Das liegt einerseits am nötigen Kompromiß zwi-

[20] Siehe z. B. [HK01].

schen Aussagekraft und Informationsmenge. Andererseits wurden Klassen aus der Ergebnismenge entfernt, wenn sie keine zusätzlichen Informationen beitrugen oder das Ergebnis undeutlich machten. Darauf wird jedoch im Text hingewiesen. Abbildung 4.5 wurde außerdem mit einer Bildbearbeitungssoftware so verändert, daß die weit auseinanderliegenden Fundstellen in zwei Klassen in den Bildausschnitt paßten. Es wurde jedoch das Ergebnis sachlich nicht verändert. Außerdem sind die verwandten Suchbegriffe immer eingeblendet, so daß die Ergebnisse nachvollzogen werden können.

Aspekt Modelldatenmanipulation

Unter diesem Begriff werden alle Handlungen verstanden, die Instanzen von Modellkonstrukten anlegen oder zerstören, bzw. Eigenschaften der Instanzen bearbeiten.

Begründung In den Untersuchungen des Quellcodes wurde deutlich, daß der Code zur Manipulation von Modelldaten meist direkt in die Methoden eingefügt wurde, die zur Behandlung von Events in der Swing-Bibliothek vorgesehen sind (z. B. `actionPerformed`-Methoden). Es ist daher wahrscheinlich (bzw. war schon bekannt), daß Code zur Modelldatenmanipulation über die gesamte Anwendung verstreut ist. Da verschiedene GUI-Elemente gleiche Aktionen auslösen können, ist außerdem mit dupliziertem Code zu rechnen.

Erkennung Die Klasse `Tessi.DRModel` stellt Methoden zum Erzeugen und Löschen von einigen Modellelementen bereit. Deren Aufruf kann zur Erkennung von Beginn und Ende des Lebens von Instanzen benutzt werden.

Modellelemente, die fest zu einem anderen Modellelement gehören, können über Methoden des besitzenden Modellelementes erzeugt werden. Beispiele dafür sind die Metamodellelemente `Attribute` und `Method`, für deren Erzeugung Methoden in `Tessi.Datatypes.Class` vorgesehen sind. In der Regel werden diese Methoden benutzt, es gibt aber auch Ausnahmen. Weitere Eigenschaften der Modellelemente werden durch `set`-Methoden beeinflußt. Der Aufruf solcher Methoden kann zur Erkennung der Bearbeitung einer Instanz des Modellkonstruktes Klasse verwendet werden.

Von der Betrachtung ausgeschlossen wurden die Klasse `Tessi.DRModel` und alle Klassen des Packages `Tessi.Datatypes`. Diese wurden als Ansatz eines Moduls angesehen, welches die Modellmanipulation kapselt.

Analyse und Ergebnis Die Suche über alle benutzten Metamodellelemente zugleich würde sehr unübersichtliche Resultate in AMT liefern und wurde daher nur für jeweils ein solches Element durchgeführt. Abbildung 4.3 zeigt einen Ausschnitt des graphischen Ergebnisses für Klassen.

Man kann sehen, daß vor allem die Bearbeitung von Eigenschaften auf viele Klassen und Methoden verteilt ist. Bei genauerer Betrachtung fällt auch auf, daß eine wesentliche Quelle der Zersplitterung und Codeduplizität die Tatsache ist, daß in den sogenannten Expansionsdialogen direkt das Modell bearbeitet wird (von den 13 gefundenen Klassen mit relevantem Code implementieren 7 Expansionsdialoge).

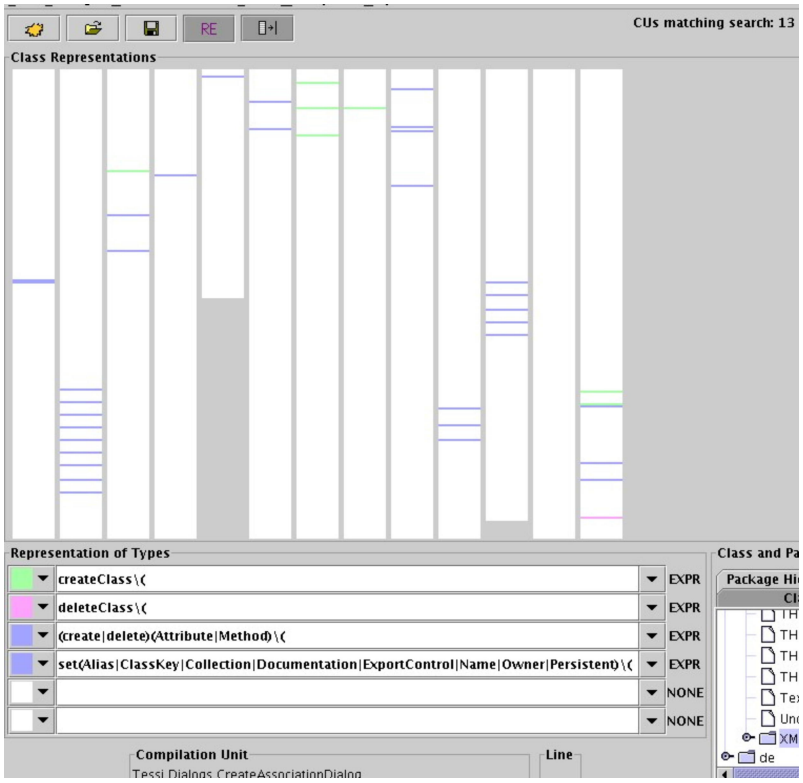

Abb. 4.3: Untersuchung zur Verteilung von Codefragmenten zur Manipulation von Klassen in TESSI 1.1

Aspekt Modelldatenvisualisierung

Die Visualisierung von Modelldaten umfaßt drei Bereiche: die Hervorhebung von Namen im Spezifikationstext, die Anzeige von Ausschnitten des Modells im Rahmen der Modelldatenmanipulation und die Darstellung als Baum.

Begründung Schon durch die Aufteilung in drei Gruppen ist eine starke Zersplitterung zu vermuten. Dies ist aber auch der Fall, wenn man die Teile einzeln betrachtet.

Erkennung Zur Erkennung können die `get`-Methoden benutzt werden, mit denen Eigenschaftswerte von Modellelementen abgefragt werden. Um die Suche zu präzisieren, muß auch der Typ der Modellelemente einbezogen werden. Interessant sind Fundstellen, die Treffer für beide Suchparameter aufweisen. Die Klassen `Tessi.DRModel` und `Tessi.DRText` sowie Klassen aus dem Package `Tessi.Datatypes` werden nicht betrachtet.

Analyse und Ergebnis Auch in diesem Fall liefert die AMT-Suche über alle Modellelementetypen zugleich ein unübersichtliches Ergebnis. Die Suche wurde nach Typ und der Verwendung von `get`-Methoden durchgeführt. Aus der gefun-

denen Menge wurden alle Klassen entfernt, die keine Anzeige implementieren. Eingeschlossen wurden allerdings die Hilfsdatentypen, deren einziger Zweck die Unterstützung der Anzeige ist.

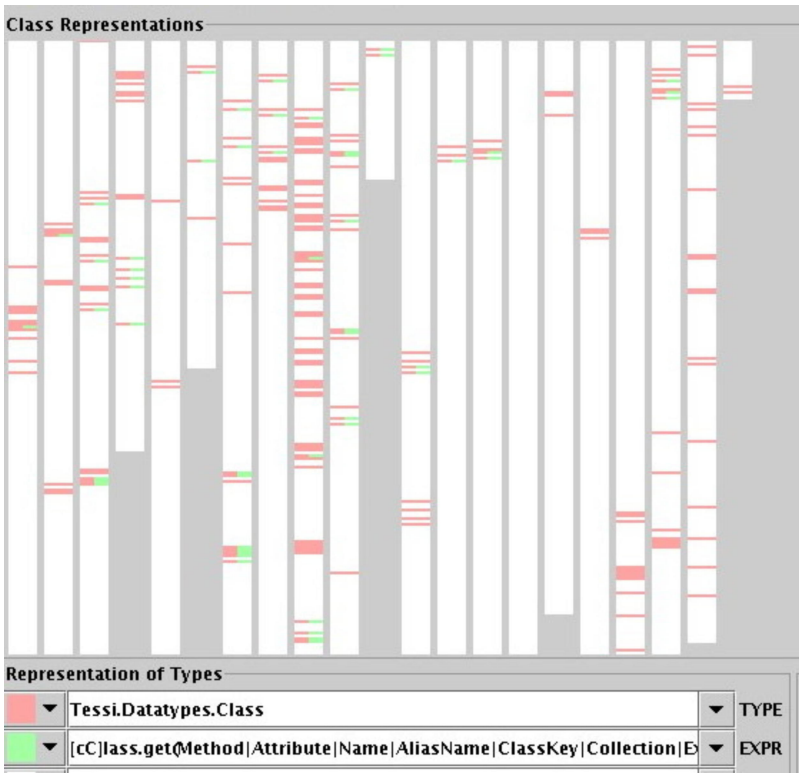

Abb. 4.4: Verteilung von Codefragmenten zur Abfrage von Eigenschaften von Klassen in TESSI 1.1

In Abbildung 4.4 ist das Ergebnis zu Klassen zu sehen. Die Ansicht soll Codefragmente visualisieren, die zur Anzeige von Eigenschaften von Instanzen des Metamodellelementes Klasse in Dialogen dienen. Gezeigt ist also das Ergebnis für ein Metamodellelement bezüglich einer der drei erwähnten Gruppen von Visualisierungen.

Die Zersplitterung ist vor allem an den Zeilen deutlich zu erkennen, die mit mehreren Farben gekennzeichnet sind. Interessant sind auch Stellen, an denen beide Farben nah beieinander vertreten sind. Sequenzen „interessanter“ Zeilen zeigen Stellen, an denen eine Eingabenmaske „gefüllt“ wird.

Betrachtet man den Code im Detail, so fällt auf, daß graphische Komponenten in großer Zahl und sehr häufig als *inner classes* definiert wurden. In den verschiedenen Eingabedialogen und -masken findet Wiederverwendung von Code fast nicht statt. Nur einige erweiterte Standard-Swingkomponenten werden an verschiedenen Stellen verwendet. Die drei für die Eingabe von Modellelementen und Eigenschaften primär gedachten Masken `Tessi.StaticStructureToolBar`, `Tessi.SequenceDiagramToolBar` und `Tessi.StateDiagramToolBar` definieren z. B. für jedes Metamodellelement eine Struktur graphischer Komponenten. Sie

werden sämtlich von inneren Klassen implementiert. Die Struktur, Bezeichnung und Teile des Codes sind jeweils sehr ähnlich. Wie bereits angedeutet, gibt es dennoch keine Wiederverwendung (z. B. durch Vererbung oder Komposition). Man kann hier von einer großen Menge duplizierten Codes ausgehen. Die genannten Klassen gehören folgerichtig zu den größten Klassen von TESSI 1.1, was sowohl LOC, Attributanzahl oder Methodenanzahl betrifft. `Tessi.StaticStructureToolBar` ist mit über 2000 Zeilen auch die mit Abstand größte Klasse.

Dies alles trägt wesentlich zur Zersplitterung des Codes bei, ohne daß es eigentlich von der Architektur her nötig wäre. Würden die drei genannten Klassen *refactored*, ergäbe sich bereits eine drastische Ausdünnung der Fundstellen in Abbildung 4.4.

Ein weiterer Faktor ist die Duplizität des darstellenden Codes in den Expansionsdialogen. Sie machen ebensowenig von Wiederverwendung Gebrauch. Etwa die Hälfte der betroffenen Klassen implementieren Expansionsdialoge. Auch hier könnte die Struktur von einer sinnvollen Auslagerung von Komponenten stark profitieren.

Für die Hervorhebungen im Text und die Baumdarstellung ergibt sich, daß der „arbeitende“ Code auf wenige Methoden konzentriert ist, die den Klassen `Tessi.DRText` und `Tessi.DRTree` angehören. Die Ansteuerung dieses Codes als Reaktion auf Ereignisse ist jedoch verstreut.

Aspekt Textgenerierung

Unter Textgenerierung wird die Erzeugung von Texten in verschiedenen Formaten aus den vorhandenen Modelldaten verstanden. Die Erzeugung von Metriken wird, obwohl sie gleichfalls einen Text generiert, von dieser Untersuchung ausgenommen.

Begründung Bei der Analyse des Quellcodes wurde festgestellt, daß ein Großteil des Codes in der Klasse `Tessi.DRText` der Erzeugung von Texten dient. Die erzeugenden Methoden sind, mit einer Ausnahme, ausschließlich `private`. Das führt zu der Annahme, daß hier ein separierter *concern* vorliegt.

Erkennung Methoden, die in ihrem Code Text erzeugen, folgen einem Namensmuster. Der Name beginnt mit `generate`. Danach folgt ein Begriff, der angibt, für welches Metamodellelement Text generiert wird. Zum Schluß folgen die Suffixe `Code` oder `Text`. Sie geben an, ob es sich beim erzeugten Text um Quelltext oder eben Text handelt.

Für jede Art von Text gibt es Methoden, die das Durchlaufen der erfaßten Modellelemente initiieren und rekursiv die eben beschriebenen `generate`-Methoden aufrufen. Sie haben ebenfalls das Präfix `generate`. Der Rest des Namens besteht aus einer Bezeichnung des erzeugten Textes.

All diese Methoden sind `private`, eine Suche nach ihnen würde nur ihre Konzentration in der Klasse `Tessi.DRText` zeigen.

Die Methode `generateTexts` ist dagegen `public` und bewirkt die Erzeugung aller Texte. Mit ihrer Hilfe kann untersucht werden, an welchen Stellen die Textgenerierung angestoßen wird.

Zugriff auf bestimmte Texte erhält man durch Aufruf entsprechender Getter-Methoden. Diese bewirken jeweils die Erzeugung einer Textart. Für jede dieser Methoden existiert ebenfalls eine Methode, die statt eines Strings das Dokument, d. h. eine Instanz von `Tessi.UndoableStyledDocument` zurückliefert. Beide Gruppen sind `public` und können zur Auffindung von Zugriffen auf einzelne Textarten benutzt werden.

Analyse und Ergebnis Das Ergebnis der mit den genannten Parametern durchgeführten Suche ist in Abbildung 4.5 zu sehen.

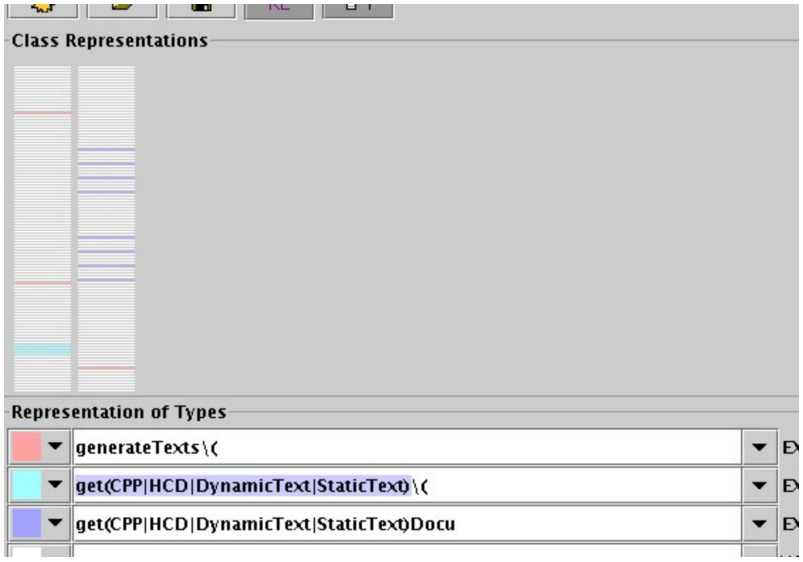

Abb. 4.5: Untersuchung zur Verteilung von Codefragmenten zur Textgenerierung in TESSI 1.1 (Die Positionen der Fundstellen sind zur besseren Ansicht per Bildbearbeitung relativ zueinander verschoben worden.)

Der gesuchte Code verteilt sich auf Fundstellen in zwei Klassen. Die linke Säule im Bild stellt die Klasse `Tessi.MainFrame`, die rechte Säule die Klasse `Tessi.TextPanel` dar.

In `Tessi.MainFrame` werden viele globale Steuerungsaufgaben in einer relativ großen Methode[21] abgehandelt. Hier wird auch das Erzeugen und Auslesen der Texte beim Laden bzw. Speichern von Projekten veranlaßt.

Die Klasse `Tessi.TextPanel` ist eine graphische Komponente von TESSI 1.1, die die Anzeige aller Texte, des Editors, der Hilfe und der Legende koordiniert. Aufrufe an die Textgenerierung finden im Konstruktor und in einer `updateView`-Methode statt. Diese Methode wiederum wird ausschließlich innerhalb von `Tessi.MainFrame` aufgerufen, und zwar fast nur in Reaktion auf Ereignisse, die in `actionPerfomed` behandelt werden.

Zusammenfassend kann man sagen, daß der untersuchte *concern* fast vollständig separiert ist. Die Ausnahme bildet der Aufrufcode als Reaktion auf

[21] net.sourceforge.Metrics ermittelt 262 LOC und eine McCabe-Komplexität (CCN) von 69. Zur Erinnerung, in [Kro97, S. 233] ist nachzulesen, daß eine CCN ab 10 erschwerte Wartung bedeutet. Die große Zahl rührt von einer langen `if-else if`-Kette her.

Ereignisse, der sich jedoch durch eine Umstrukturierung leicht separieren ließe. Ursache der Zersplitterung ist die enge Kopplung mit der GUI.

Als kritisch wird die Vermischung des Codes zur Erzeugung von Text mit Code zu anderen *concerns* gesehen, da Aufgaben sehr unterschiedlicher Art zusammen erledigt werden.

Als weiterer Aspekt wurden die Spezifikationstextbehandlung und Daten-I/O betrachtet. Wie in allen anderen Fällen ergab sich auch hier eine Zersplitterung des Steuercodes und dessen Vermischung mit Code zur Implementierung der GUI-Logik.

Da sich die Spezifikationstextbehandlung im Wesentlichen auf vorgefertigte Java-Komponenten verläßt, war sie relativ in sich abgeschlossen. Eine Ausnahme stellte hier die Erzeugung der Dokumenteinstanzen dar, die im Rahmen der Textgenerierung stattfindet. Das Analyseergebnis ist mit der Untersuchung zur Textgenerierung vergleichbar.

Unter dem Begriff Daten-I/O wurden alle Handlungen zusammengefaßt, die das Laden und Speichern von Daten durchführen. Von TESSI 1.1 bearbeitete Daten sind Spezifikationstext, generierte Texte, Alia und Modelldaten.

TESSI 1.1 speichert Modelldaten in XML-Dateien. Die Erzeugung und das Lesen dieser Dateien implementieren Klassen aus dem Package `Tessi.XML` und Unterpackages. Wie bereits deutlich wurde, kann man hier von einem Modul ausgehen.

Das Lesen und Schreiben der übrigen Daten wird von Methoden in der Klasse `Tessi.MainFrame` ausgeführt. Deren Aufruf erfolgt in der bereits erwähnten Nachrichtenroutine `Tessi.MainFrame.actionPerformed`.

Im Fall von Daten-I/O kann man ebenfalls von einem Aspekt sprechen, der teilweise gut abgetrennt ist. Teilweise ist dazugehöriger Code jedoch ungünstig platziert, kann aber ohne großen Aufwand ebenfalls separiert werden.

4.4 Zielsetzung

Als Verfeinerung der Aufgabenstellung auf Seite 2 und als Ergebnis der Untersuchungen des IST-Standes wird in diesem Abschnitt das Ziel für diese Arbeit formuliert.

4.4.1 Architektur

Die in Abschnitt 4.3.2 erarbeitete Modulstruktur teilt TESSI in funktionell zusammengehörende Bereiche. Die gewählte Aufteilung richtet sich aber auch nach den zu erwartenden Veränderungen an TESSI, besonders nach bereits geplanten Projekten. Die in TESSI 2.0 zu realisierenden Module sollten sich nach dieser Struktur richten.

Neben den Kernmodulen wird es Module geben, die den aktuellen Funktionsumfang erweitern. Es soll möglich sein, solche Erweiterungen ohne Codeveränderungen und Neukompilierung in TESSI 2.0 zu integrieren. Das gleiche gilt natürlich für Module, die als verbesserte Versionen bereits existierende Module ersetzen sollen.

Es ist zu berücksichtigen, daß erweiternde oder verbesserte Module auch Zugang zu Mechanismen zur Nachrichtenerzeugung und -verteilung haben müssen.

Es muß also ein Verteilungssystem für Nachrichten geben, welches von den Modulen benutzt und auch erweitert werden kann.

4.4.2 Implementierung

Die entworfene Architektur ist zu implementieren. Der Funktionsumfang von TESSI 1.1 ist in TESSI 2.0 wieder zu erreichen. Die entstehenden Implementierungen der Module können als Standardlösung mit dem Funktionsumfang der spezifizierten Module gesehen werden.

Bei der Implementierung ist darauf zu achten, daß möglichst viel TESSI 1.1-Code wiederverwendet wird. Die Datenhaltung soll unter Verwendung eines Repositorys eines Fremdherstellers realisiert werden. Ausgewählt dafür wurde *Metadata Repository* (MDR) aus dem Netbeans Projekt.

4.4.3 Dokumentation

Die geleistete Arbeit ist zu dokumentieren, wobei die Dokumentation sich an den zu erwarteten Nutzergruppen orientieren muß. Dies sind einerseits Anwender des Programmes zu Analysezwecken. Dafür ist ein Nutzerhandbuch vorzusehen. Andererseits wird TESSI 2.0 Gegenstand von verbessernden und erweiternden Projekten sein. Für diese Benutzer ist ein Programmierhandbuch und eine API-Dokumentation vorzusehen.

4.4.4 Auswertung

Die Arbeit ist auszuwerten. Vor allem ist zu untersuchen, welche Vor- und Nachteile sich aus dem Übergang von TESSI 1.1 zu 2.0 ergeben.

5. BESCHREIBUNG DER NEUEN ARCHITEKTUR

Dieses Kapitel beschreibt die Architektur von TESSI 2.0. Es dient vor allem der Spezifizierung der Aufgaben aller Module sowie der Dienste, die die Kommunikation zwischen den Modulen sicherstellen. Die Implementierung von TESSI 2.0 wird als Standardimplementierung der Spezifikation aus diesem Kapitel gesehen. Es wird erwartet, daß spätere Projekte Funktionen hinzufügen und Verbesserungen durchführen werden, dabei allerdings die Struktur beibehalten. Die Standardimplementierung wird in Kapiel 6 betrachtet.

Die Beschreibung der Architektur orientiert sich an der Modulstruktur der neuen Version. Abschnitt 5.1 gibt zunächst einen Überblick der Struktur und erläutert gemeinsame Eigenschaften. Die Abschnitte 5.2 bis 5.9 beschreiben jeweils ein Modul. Zu jedem Modul werden aufgeführt:

- eine informelle Beschreibung der Aufgaben des Moduls
- eine Übersicht der Nachrichten, die das Modul emittiert
- eine Übersicht der Nachrichten, auf die das Modul reagiert; dazu kommt die Beschreibung der Reaktion
- eine Beschreibung der benötigten graphischen Komponenten

Abschnitt 5.10 schließlich beschreibt eine Bibliothek von Adapterklassen, mit deren Hilfe der Zugriff auf die Modelldaten in der Datenhaltung vereinfacht wird.

5.1 Allgemeine Struktur und allgemeine Dienste

Beim Entwurf von TESSI 2.0 wurde von der in Abschnitt 4.3.2 erarbeiteten Modulstruktur (siehe Abbildung 4.2) ausgegangen. Die tatsächliche Struktur erläutert Abschnitt 5.1.1.

TESSI ist ein System mit einer graphischen Nutzerschnittstelle (GUI). Viele Eckpunkte einer solchen Oberfläche sind durch die *de facto*-Standards gängiger graphischer Systeme vorgegeben. Zum Beispiel sind einige Menüs, Befehle und Tastaturkürzel in fast allen Programmen mit einer GUI wiederzufinden. Dieser Anteil an der GUI wird hier als gemeinsame Oberfläche bezeichnet und durch ein Basismodul realisiert. Während der Abschnitt 5.2 detaillierter auf dieses Modul eingeht, sollen die dort angesiedelten Mechanismen vorher genauer betrachtet werden.

Zwei solche Mechanismen bzw. Aufgaben sind die Handhabung (also das Laden und Einbinden) von Modulen (Abschnitt 5.1.2) und das bereits angedeutete Nachrichtensystem (Abschnitt 5.1.3). Damit Module zur gemeinsamen Oberfläche beitragen können, muß eine Funktion existieren, die das Hinzufügen von

graphischen Komponenten durch Module gestattet. Sie sollte die Möglichkeiten der Module nicht unnötig beschränken, allerdings sollte sie auch so einfach wie möglich sein. Diese Funktion wird in Abschnitt 5.1.4 vorgestellt. Schließlich kann sich auch die Situation ergeben, daß ein Modul Steuerungsmöglichkeiten direkt aus der gemeinsamen Oberfläche benötigt. Für solche Zwecke ist ebenfalls ein entsprechender Mechanismus nötig (Abschnitt 5.1.5).

5.1.1 *Verfeinerte Modulstruktur*

Die Anforderungen an eine neue Architektur führten zu einer Modulstruktur. Der Funktionsumfang von TESSI 2.0 war durch die Fähigkeiten von TESSI 1.1 festgelegt. Kriterien für die Aufteilung in Module waren die logische Zusammengehörigkeit der Funktionen (siehe Abschnitt 4.3) und erwartete Änderungen in der Zukunft (siehe Abschnitt 4.2).

Für die Module galten zusätzliche Anforderungen. Sie sollten ohne Veränderung und Rekompilierung der Restanwendung austauschbar sein und Steuerungsmöglichkeiten durch ein Verteilsystem von Nachrichten innerhalb von TESSI 2.0 besitzen (siehe Abschnitt 4.4).

Unter den gefundenen Modulen ist aber keines, daß zusätzlich notwendige Mechanismen aufnehmen könnte, die für alle anderen Module gleich wichtig sind. Dazu gehören das Nachrichtensystem und eine Funktion, welche die vorhandenen Module auffindet, lädt sowie Anbieter und Klienten von Diensten zueinander bringt. Diese Funktionen passen logisch in kein anderes Modul. Für diese Aufgaben wurde also ein weiteres Modul vorgesehen (Modul „Basis“). Dazu kommen Eingriffsmöglichkeiten in die gemeinsame Oberfläche. Die tatsächliche Modulstruktur von TESSI 2.0 zeigt Abbildung 5.1.

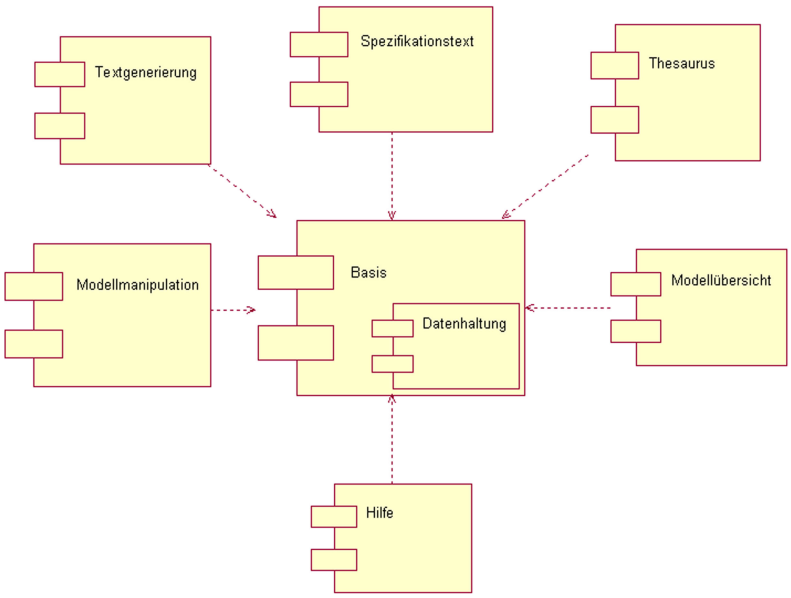

Abb. 5.1: Struktur der Module in TESSI 2.0.

Die Beziehungen der Module untereinander beschränken sich in TESSI 2.0

demzufolge auf insgesamt vier Dienste: Modulhandhabung, Nachrichtensystem, Datenhaltung und GUI-Erweiterung. Alle Module sind direkt nur vom Basismodul abhängig.

Das Modul „Datenhaltung“ ist jedoch ein Spezialfall. Es ist das einzige Modul, welches einen Dienst exportiert. Dieser Fall wird gesondert behandelt (siehe Abschnitte 5.1.2 und 6.2).

5.1.2 Modulhandhabung

TESSI 2.0 ist aus acht Modulen aufgebaut. Die Funktionen des Programms sind auf diese Module verteilt. Zu einem sinnvollen Ablauf von TESSI 2.0 müssen alle Module vorhanden sein und geladen werden sowie die benötigten Importe und Exporte der Module bedient werden.[1] Diese Aufgabe wird von der in diesem Abschnitt beschriebenen Funktion realisiert.

Konkret sind zur Realisierung dieser Funktion folgende Rahmenbedingungen nötig:

- Definition eines „Ortes“, an dem die vorhandenen Module gesucht, bzw. gefunden werden; eine einfache Möglichkeit ist ein Verzeichnis im Dateisystem (Modulverzeichnis)
- Definition eines Transportformates für Module (z. B. eine gepackte Datei)
- eine Instanz, die diesen Ort prüft und das Vorhandensein von Modulen feststellt
- Definition einer Beschreibung für importierte und exportierte Dienste eines Moduls (z. B. eine Datei mit entsprechendem Format)
- eine Instanz, die die Beschreibungen ausliest und auswertet
- eine Instanz, die die Registrierung exportierter Dienste in der Applikation, bzw. importierter Dienste im Modul vornimmt
- eine Instanz, die nach dem Laden aller Module feststellt, ob die Programmfunktion gewährleistet ist

Daran erkennbar sind die Einzelhandlungen, die zur Realisierung ausgeführt werden müssen:

1. Durchsuchen des Modulverzeichnisses nach vorhandenen Modulen
2. Laden der Module und der Beschreibung der Moduldienste (Import und Export)
3. Registrieren der vom Modul exportierten Dienste
4. Versorgen des Moduls mit Informationen zu importierten Diensten
5. Initialisierung des Moduls

[1] Einige Module müssen in jedem Fall vorhanden sein (Basis, Datenhaltung). Andere sollten vorhanden sein, TESSI kann jedoch in großen Teilen auch ohne sie funktionieren (Thesaurus, Hilfe). Es müssen also nicht „alle“, aber mindestens die „wichtigen“ Module vorhanden sein.

6. (nachdem alle Module geladen sind) Feststellung, ob die minimal benötigten Dienste vorhanden sind; Auslösung einer angemessenen Reaktion

Die Struktur der Instanzen, die als Lösung des Problems in TESSI 2.0 definiert wurde, zeigt Abbildung 5.2.

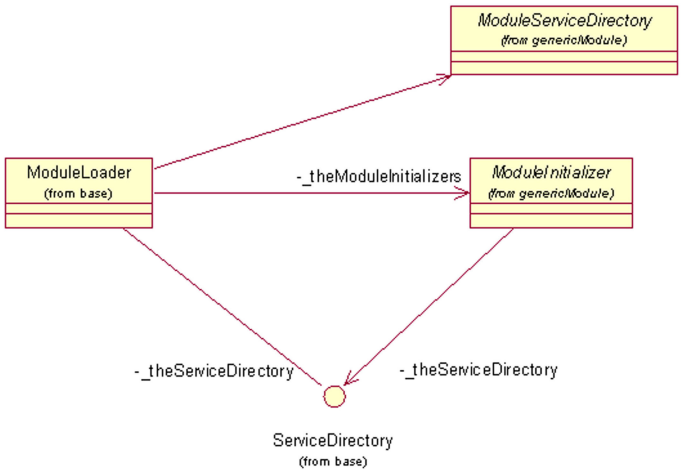

Abb. 5.2: Laden und Initialisierung von Modulen in TESSI 2.0

Das `ServiceDirectory` dient als Verzeichnis der grundlegenden Dienste, die alle Module importieren. Der `ModuleLoader` lädt nacheinander alle Module und die Beschreibungen der Dienste. Das konkrete Transport- und Beschreibungsformat ist hier noch nicht von Bedeutung. Es sind abstrakte Klassen definiert, die den Export der Moduldienste unterstützen (Klasse `ModuleServiceDirectory`) oder den Import von Diensten und die Modulinitialisierung durchführen (Klasse `ModuleInitializer`). Jedes Modul muß Implementierungen für diese Klassen bereitstellen. Der `ModuleLoader` stellt die implementierenden Klassen fest, erzeugt sie und ruft entsprechende Methoden auf, die die erwähnten Aufgaben erledigen.

Nach dem Laden der Module muß festgestellt werden, ob der Ablauf des Programmes möglich ist. Das hängt davon ab, ob zumindest die unbedingt benötigten Dienste vorhanden sind. Welche Dienste sind dies? Alle Module benötigen zunächst die grundlegenden Dienste, die in diesem Abschnitt sowie in den Abschnitten 5.1.3, 5.1.4 und 5.1.5 betrachtet werden. Diese Dienste werden vom Basismodul bereitgestellt, deren Vorhandensein kann zum Zeitpunkt des Ladens der Module als gegeben vorausgesetzt werden (der `ModuleLoader ist selbst Teil dieses Moduls`). Der einzige grundsätzlich wichtige Dienst, der von einem anderen Modul realisiert wird, ist die Datenhaltung. Es muß also lediglich geprüft werden, ob ein — und zwar genau ein — solches Modul vorhanden ist.

Details zur Implementierung sind im Abschnitt 6.2 sowie außerdem in der API-Dokumentation erläutert.

5.1.3 Nachrichtensystem

TESSI 2.0 verfügt über eine graphische Oberfläche. Nutzereingaben können an verschiedenen Stellen vorgenommen werden und sollen teilweise die gleiche Reaktion hervorrufen (z. B. der Menübefehl Bearbeiten→Suchen und das Tastenkürzel Strg-F). Außerdem befinden sich der Empfänger einer Nutzereingabe und der Verarbeiter nicht unbedingt im selben Modul. Um die nötige Kommunikation zwischen beiden zu gewährleisten, gibt es verschiedene Möglichkeiten.

Eine graphische Bibliothek wie Java Swing definiert viele Beziehungen, die der Realisierung dieser Kommunikation dienen. Sie beruhen in der Regel auf dem *Listener*-Konzept.[2] Eine Instanz, nämlich der *Listener*, der auf ein Ereignis reagieren soll, meldet sich bei der Instanz an, die das Ereignis registriert. Tritt das Ereignis auf, wird der *Listener* über eine definierte Schnittstelle benachrichtigt, d. h. eine bestimmte Methode der *Listener*-Klasse wird aufgerufen. Dieses System hat den Vorteil, daß es fertig zur Verwendung bereit steht. Nachteile sind die Ausrichtung auf die Benutzeroberfläche und die vielen Variationen der Beziehung.[3] Außerdem müssen sich Quelle und Senke der Nachrichten direkt kennen[4], damit die Anmeldung des *Listeners* sowie später die Kommunikation zustande kommen können.

Für TESSI 2.0 sollte ein Nachrichtensystem entworfen werden, welches die genannten Nachteile nicht hat. Abbildung 5.3 zeigt die gewählte Lösung.

Zentraler Teil des Nachrichtensystems ist das `Command`, worunter ein beliebiger Befehl zu verstehen ist. Ein `Command` verbindet ein Ereignis (Klasse `Event`), als auslösendes Ereignis, und eine Anzahl von Aktionen (Klasse `Action`), die als Reaktion auf das Ereignis abgearbeitet werden. Ein `Command` dient als Mittel, um die Bearbeitung der Handlung zu veranlassen. Dazu wird von einem Klienten eine Instanz des `Event`s erzeugt und dem dazugehörenden `Command`[5] zur Bearbeitung übergeben. Aus der Sicht der Ereignisquelle wird also ein Befehl gegeben, dem eine Instanz des Ereignisses als Parameter zugeordnet wird.

Die Struktur aus Abbildung 5.3 wurde vom *Command-Design Pattern*[6] abgeleitet. Der Unterschied besteht darin, daß die Handlung des `Command` des Entwurfsmusters herausgelöst wurde, um an einen Befehl mehrere Handlungen binden zu können. Dazu kam noch die `CommandRegistry`, die verantwortlich ist für die Registrierung von `Event`s und `Action`s und die Verwaltung von `Command`s.

Aktionen werden von den Modulen für bestimmte Ereignisse registriert. Die Registrierung einer Aktion für ein Ereignis hat zur Folge, daß beim Auftreten des Ereignisses die registrierte Aktion ausgeführt wird. Module können Ereignisse auch ohne Angabe einer `Action` registrieren. Damit macht ein Modul die Existenz eines Ereignisses bekannt. Diese Handlung entspricht der Absicht, nur die Quelle des Ereignisses zu sein (dazu wird das dazugehörende `Command` benötigt).

[2] Obwohl es in Swing einen anderen Namen trägt, entspricht das *Listener*-Konzept dem bekannten Beobachter-*Design Pattern* (engl. *Observer*). Siehe [GoF96, S. 287 ff].

[3] Eine Suche nach Klassen und Interfaces mit dem Namensteil „Listener“ liefert in den Packages von AWT und Swing 48 Treffer (Java SDK 1.4.1).

[4] Natürlich kann man sich hier einen Indirektionsmechanismus ausdenken, um eine Assoziation der Kommunikationspartner zu umgehen. Dies ist jedoch eine Veränderung des ursprünglichen Designs.

[5] Es wird sichergestellt, daß zu einer bestimmten Ereignisklasse immer genau ein `Command` gehört. Voraussetzung ist jedoch, daß die Ereignisklasse vorher mindestens einmal bei einer Registrierung benutzt wurde.

[6] Siehe [GoF96, S. 273 ff.].

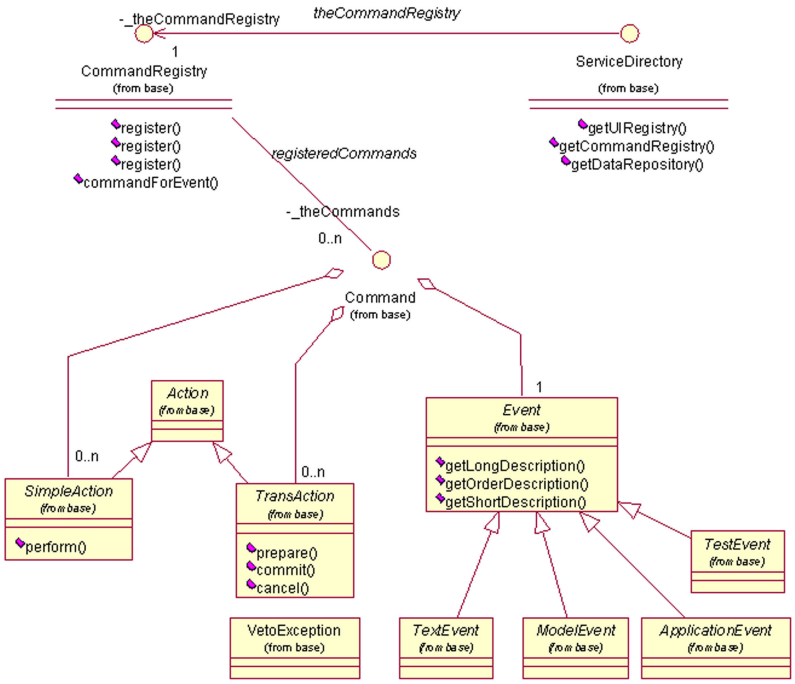

Abb. 5.3: Nachrichtensystem in TESSI 2.0

Eine Konsequenz davon ist, daß registrierte Ereignisse, für die kein Modul eine `Action` registriert, unbearbeitet bleiben.

Alle in TESSI 2.0 auftretenden Ereignisse sind von der Oberklasse `Event` abzuleiten. Als direkte Kinder sind vier Unterklassen definiert, die als Oberklassen für Kategorien von Ereignissen dienen. Die vier Kategorien sind:

- Globale Steuerung der Applikation (Klasse `ApplicationEvent`); betrifft alle Ereignisse, die den Lebenszyklus der Applikation bzw. des bearbeiteten Projektes betreffen
- Ereignisse, die die Bearbeitung von Texten betreffen (Klasse `TextEvent`)
- Ereignisse, die die Bearbeitung von Modellelementen und deren Eigenschaften betreffen (Klasse `ModelEvent`)
- Testereignisse (Klasse `TestEvent`); betrifft Ereignisse, die zum Testen bestimmten Verhaltens von Modulen verwendet werden können (nicht zur normalen Funktion von TESSI)

Beim Auftreten eines Ereignisses wird in TESSI 2.0 immer die Klasse des Ereignisses betrachtet, d. h. es werden das konkrete, aufgetretene Ereignis sowie alle Oberklassen des Ereignisses bearbeitet. Dadurch ist es möglich, daß Aktionen auf Kategorien von Ereignissen reagieren, statt nur auf spezifische Ereignisse[7].

[7] Die Reaktion auf spezifische Ereignisse ist natürlich dennoch möglich.

Module können durch Subklassenbildung das Nachrichtensystem „dezentral“ erweitern. Es sind dazu nur von `Event` abgeleitete Klassen nötig. Sollen andere Module auf solche Ereignisse reagieren können, muß deren Existenz allerdings auf anderem Weg bekannt gemacht werden. Eine Möglichkeit, zur Laufzeit festzustellen, welche zusätzlichen Ereignisklassen von Modulen definiert werden, ist nicht vorgesehen. Es ist natürlich weiterhin möglich, auf Subklassen bekannter Ereignisse zu reagieren.

Analog zu anderen Nachrichtensystemen, wie zum Beispiel in Java Swing, können Ereignisse zusätzliche Informationen (Parameter) zum aufgetretenen Ereignis in sich tragen. Für die in TESSI 2.0 definierten Ereignisse ist das teilweise der Fall.

Die Behandlung der Nachrichten wird von Aktionen ausgeführt. Superklasse für alle Aktionen ist die Klasse `Action`. Interessanter sind jedoch die beiden Kindklassen `SimpleAction` und `TransAction`. Sie spiegeln verschiedene Aspekte bei der Bearbeitung von Ereignissen wider.

Die Klasse `SimpleAction` repräsentiert die einfache Reaktion auf eine Benachrichtigung. Im Fall des Auftretens des gewünschten Ereignisses wird die einzige definierte Methode `perform` aufgerufen. Implementierungen verarbeiten hier das Ereignis.

Die Klasse `TransAction` repräsentiert dagegen eine Handlung, die nur unter bestimmten Umständen auf das Ereignis reagiert. Es besteht sogar die Möglichkeit, die Bearbeitung des `Event`s insgesamt zu blockieren (ein Veto einzulegen). Die Ereignisbehandlung wird zweistufig durchgeführt:

1. Für alle registrierten `TransAction`s wird die Methode `prepare` aufgerufen. In dieser Methode müssen Implementierungen prüfen, ob die Bearbeitung des Ereignisses aus ihrer Sicht durchgeführt werden kann oder sollte. Existieren Bedingungen, die die weitere Behandlung des `Event`s nicht zulassen, muß ein Veto eingelegt werden. Dies ist durch das Werfen einer Ausnahme (Klasse `VetoException`) realisiert.

2. (a) Stellt das Ausführungssystem kein Veto fest, wird für alle registrierten Aktionen die Methode `commit` aufgerufen. Implementierungen führen hier die Handlung durch oder schließen sie ab.

   (b) Tritt in der ersten Phase ein Veto auf, wird für alle registrierten Aktionen die Methode `cancel` aufgerufen. Implementierungen machen hier alle Veränderungen rückgängig, die in der ersten Phase eventuell durchgeführt wurden.

Der Term „alle registrierten Aktionen“ meint Aktionen, die für das aufgetretene Ereignis oder für eine Oberklasse dieses Ereignisses registriert sind. Für das Blockieren ist es unerheblich, für welches `Event` die `TransAction` registriert ist, die das Veto abgegeben hat. Eine `TransAction` für eine Oberklasse des aufgetretenen Ereignisses kann die Bearbeitung des Ereignisses ebenso blockieren, wie eine `TransAction` für das Ereignis selbst. Blockiert wird die Bearbeitung immer komplett, d. h. keine der für das aufgetretene Ereignis oder seine Oberklassen registrierten `SimpleAction`s oder `TransAction`s wird abgearbeitet. „Nicht abgearbeitet werden“ bedeutet für `SimpleAction`s kein Aufruf irgendeiner Methode. Für `TransAction`s bedeutet es die Aufruffolge (`prepare`, `cancel`).

Alle Aktionen müssen von einer der beiden Subklassen von `Action` abgeleitet sein. Die Registrierung von Aktionen beruht auf diesen beiden Typen.

Das Nachrichtensystem ist eines der grundlegenden Dienste, die (wahrscheinlich) von jedem Modul importiert werden. Wie in Abbildung 5.3 zu sehen, ist es daher im `ServiceDirectory` verankert und wird beim Laden den Modulen bekannt gemacht.

In TESSI 2.0 wurde eine große Zahl von Standardereignissen (sowie die entsprechenden Reaktionen) vordefiniert. Anhang A listet diese Ereignisse auf, beschreibt die Bedeutung und die Standardreaktion auf deren Auftreten.

5.1.4 Graphische Komponenten

TESSI 2.0 verfügt über eine graphische Oberfläche. Teile dieser Oberfläche werden vom Basismodul definiert (siehe dazu Abschnitt 5.2). Entsprechend ihrer Funktion kann aber für Module die Notwendigkeit bestehen, graphische Komponenten zur gemeinsamen Oberfläche beizutragen, d. h. eigene graphische Komponenten zu definieren. Bei der Initialisierung des Moduls müssen die graphischen Komponenten angemeldet werden.

Es wurde darauf verzichtet, spezifische Schnittstellen für die von Modulen definierbaren Komponenten zu entwerfen. Statt dessen wird auf Java Swing zurückgegriffen und zwar konkret auf die Klasse `javax.swing.JComponent`. Zusätzlich werden Rahmenbedingungen für die Komponenten vorgegeben und ein rudimentärer graphischer Steuerungsmechanismus bereitgestellt. Abbildung 5.4 verdeutlicht den Zusammenhang.

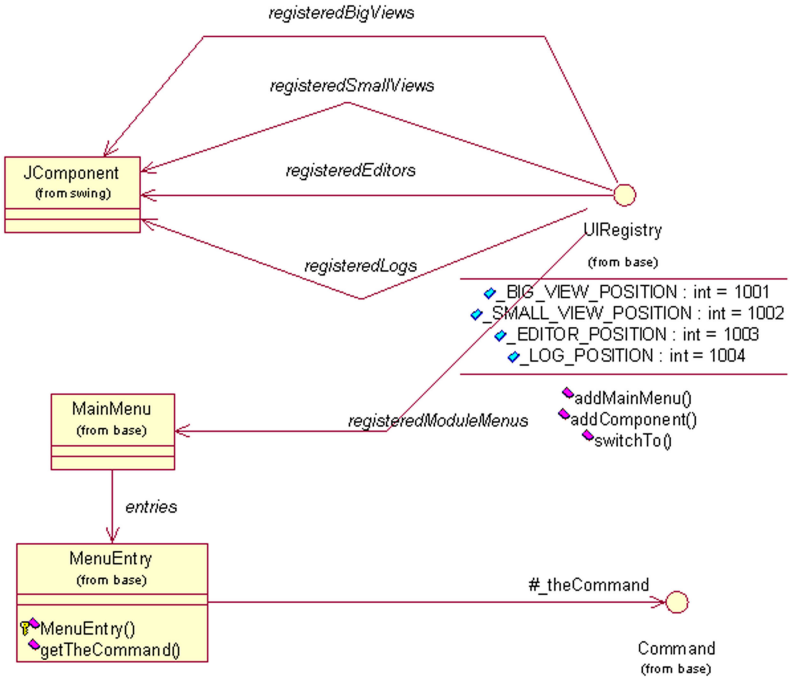

Abb. 5.4: Verwaltung graphischer Komponenten von Modulen in TESSI 2.0

Die Verwaltungsinstanz für alle graphischen Komponenten von Modulen ist die `UIRegistry`. Hier müssen Module ihre graphischen Komponenten registrie-

ren. Dafür reicht eine einzige Methode aus. Bei der Registrierung muß angegeben werden, an welcher Position in der Oberfläche die Komponenten angezeigt werden sollen. Vier solcher Positionen oder Bereiche sind definiert. Abbildung 5.5 zeigt das (fast leere) Fenster von TESSI 2.0 und verdeutlicht diese Bereiche. Jeder Bereich ist so angelegt, daß er mehrere graphische Komponenten beherbergen kann. Der Nutzer entscheidet, welche Komponente er sehen möchte. In der Abbildung ist zu sehen, daß die Komponenten über Karteireiter wählbar sind.

Den einzelnen Bereichen sind symbolische Konstanten zugeordnet, die bei der Registrierung angegeben werden müssen. Die Konstanten sind im Interface `UIRegistry` definiert. Die Namen der Konstanten wurden in Abbildung 5.5 benutzt, um die Bereiche zu beschriften.

Die Schnittstelle der `UIRegistry` bietet darüber hinaus noch eine Methode `switchTo`. Über diese Methode soll ein Modul registrierte Komponenten in den Vordergrund schalten können, wenn sie von anderen Komponenten in ihrem Bereich verdeckt sind. Das kann verwendet werden, wenn ein Modul zwei graphische Komponenten registriert hat und ein gezieltes Umschalten ermöglicht werden soll.

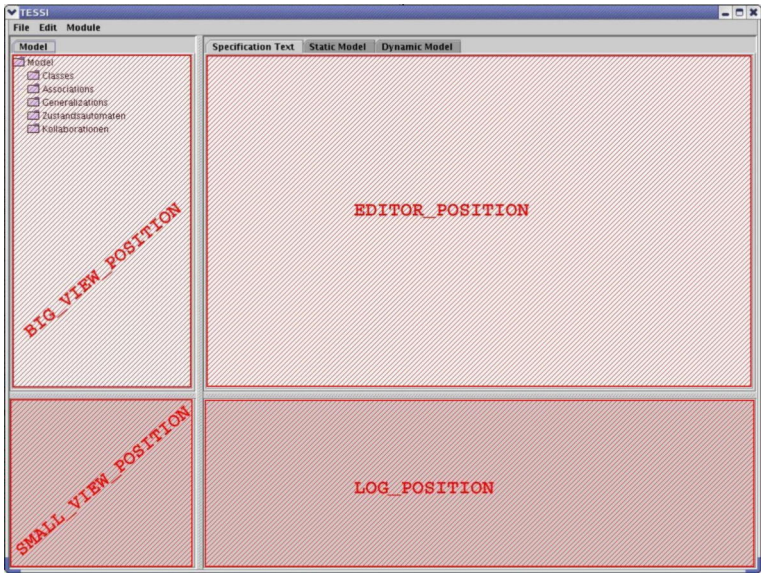

Abb. 5.5: Ansicht der Oberfläche von TESSI 2.0. Die vier Anzeigebereiche sind hervorgehoben und beschriftet.

Die Implementierung der graphischen Komponenten und der Präsentationslogik liegt völlig in der Verantwortung des jeweiligen Moduls. Graphische Komponenten könnten z. B. umfangreiche Eingabemasken darstellen. Einzige Bedingung ist, daß das gruppierende Element vom Typ `javax.swing.JComponent` ist.

Die Möglichkeiten der in diesem Abschnitt beschriebenen Funktion sind absichtlich einfach gehalten und beschränken sich auf das momentan Notwendige. Es können damit alle Aufgaben erfüllt werden, die für TESSI 2.0 zu erfüllen

waren. Es ist jedoch ohne Probleme möglich, die Schnittstelle der `UIRegistry` später zu erweitern.

5.1.5 Modulsteuerung aus der gemeinsamen Oberfläche

Gleichfalls in Abbildung 5.4 sind die Klassen zu sehen, die zur Registrierung von Steuerelementen für Module in der Oberfläche benutzt werden. Die Registrierung erfolgt wiederum durch die Module und basiert auf Menüs von `Command`s. Die für ein zu registrierendes Menü benötigten `Command`-Referenzen müssen vorher mit Hilfe der `CommandRegistry` beschafft werden.

Als gruppierendes Element für Menüeinträge ist die Klasse `MainMenu` vorgesehen. Bindeglied zwischen `MainMenu` und `Command` ist die Klasse `MenuEntry`. Sowohl `MainMenu` als auch `MenuEntry` sind in Anlehnung an das bekannte GUI-Konzept der Menüs benannt. Von Modulen registrierte Menüs können als Menüs im Hauptfenster dargestellt werden. Möglich sind jedoch auch andere Konzepte oder Kombinationen davon. Um den Unterschied deutlich zu machen, wurden daher bewußt eigene Klassen definiert und nicht entsprechende Klassen z. B. aus Java Swing benutzt.

5.1.6 Datenhaltung

Der Dienst Datenhaltung ist der einzige der vier grundlegenden Dienste in TESSI, der nicht vom Basismodul realisiert wird. Einerseits sollte dieser Dienst so gut wie möglich von der übrigen Applikation entkoppelt werden, um besser auf spätere Änderungen vorbereitet zu sein. Andererseits ist es durch JMI auch möglich, eine standardisierte Schnittstelle zumindest für die Modelldaten anzubieten. Der erhoffte Effekt ist die Unabhängigkeit von der verwendeten Software zur Verwaltung der Modelldaten.

Die Schnittstelle der Datenhaltung ist mit dem Interface `DataRepository` definiert. Die fünf zur Verfügung gestellten Methoden können zwei Gruppen von Projektdaten zugeordnet werden: dem Spezifikationstext und den Modelldaten. Beides soll im Folgenden erläutert werden.

Schnittstelle für Spezifikationstext

Zwei dieser Methoden dienen dem Zugriff auf den Spezifikationstext (die Methoden `getSpecificationText` und `setSpecificationText`). Zwei weitere Methoden sind zum Abfragen (`getIsModified`) und Setzen (`setIsModified`) eines Attributes vorgesehen, welches speichert, ob die Projektdaten seit dem letzten Abspeichern verändert wurden. Diese beiden Methoden werden ausschließlich im Zusammenhang mit dem Spezifikationstext verwendet.

Der Ablauf der Kommunikation über diese Schnittstelle ist wie folgt:

1. Der Spezifikationstext wird von der Datenhaltung bereitgestellt, d. h. es wird z. B. eine Datei geladen oder ein leerer Text erzeugt.

2. Ein Klient lädt den Text als Zeichenkette (`java.lang.String`) über einen Aufruf an die Methode `getSpecificationText`.

3. Der Text wird durch den Klienten bearbeitet. Bei Veränderungen wird durch Aufruf von `setIsModified(true)` das Modifikationsflag gesetzt.

4. Als Reaktion auf die Anforderung des Nutzers, den Text zu speichern, speichert der Klient den eventuell geänderten Text in das `DataRepository` zurück. Dies geschieht über die Methode `setSpecificationText`.

Die Schnittstelle zum Verwalten des Spezifikationstextes sowie das dazugehörige Protokoll sind an die Lösung aus TESSI 1.1 angelehnt. Die Kombination wurde aus den folgenden Gründen in TESSI 2.0 wieder verwendet:

1. Der einzige Klient für den Spezifikationstext ist momentan ein Editor (im Modul „Spezifikationstext“). Zur Implementierung des Editors wurden Klassen aus der Swing-Bibliothek eingesetzt (`javax.swing.JTextPane` als Editor, eine Subklasse von `javax.swing.text.DefaultStyledDocument` für das bearbeitete Dokument). Die Verwendung dieser Klasse hätte die Schnittstelle etwas vereinfacht, statt dessen aber eine bibliotheksspezifische, nichttriviale Schnittstelle eingebracht.

2. Planungen lassen vermuten, daß das bisherige Modell der Textrepräsentation (unstrukturierter Unicode-Text) nicht mehr ausreicht. Eine Änderung an dieser Stelle ist zu erwarten. Die Anforderungen müssen vorher untersucht werden, woraus sich eine angepaßte Schnittstelle ergeben wird.

3. Die Beibehaltung der Lösung aus TESSI 1.1 ließ den geringsten Aufwand erwarten.

Es wird hier ausdrücklich darauf hingewiesen, daß die gewählte Lösung nur unter Berücksichtigung dieser Gründe gewählt wurde. Eine endgültige, bessere Lösung, die die Erfordernisse einer anderen Textrepräsentierung berücksichtigt, muß in einem späteren Projekt noch gefunden werden. Sie sollte vor allem folgende Hauptnachteile beseitigen:

1. Aktionen des Moduls „Datenhaltung“ (das Speichern der Daten, Reaktion auf das Beenden des aktuellen Projektes durch verschiedene Ereignisse) sind von der Kooperation eines anderen Moduls abhängig (Modul „Spezifikationstext“).

2. Die Schnittstelle ist ungeeignet für die konkurrierende Bearbeitung des Spezifikationstextes durch mehr als einen Klienten.

Schnittstelle für Modelldaten

Die Schnittstelle zum Zugriff auf die Modelldaten besteht aus der Sicht des `DataRepository`s aus nur einer Methode (`getUmlPackage`). Dahinter verbirgt sich eine relativ komplexe, aus einer UML-Beschreibung generierte Struktur von Schnittstellen. Sie hat jedoch folgende Vorteile:

- Aus der Sicht des `DataRepository`s besteht die Schnittstelle zu den Modelldaten nur aus einer Methode, die außerdem auf einem unveränderlichen Teil von JMI beruht (*Reflective Interface*). An dieser Stelle sind keine Änderungen zu erwarten, außer wenn die Grundlage von JMI verändert wird.

- Die Interfaces zur UML werden direkt aus einer standardisierten Beschreibung der UML generiert. Was generiert wird, ist zur Zeit noch nicht standardisiert. Es ist jedoch zu erwarten, daß die Standardisierung der jetzigen Spezifikation bald abgeschlossen sein wird. Etwaige Veränderungen in JMI betreffen aber nur unter bestimmten Bedingungen das `DataRepository`.

Als Nachteil ist die Komplexität der Schnittstellen zu nennen, die ihren Grund in der Größe und Komplexität der UML hat und damit nicht zu ändern ist. Ein weiterer möglicher Nachteil ist die noch ausstehende Standardisierung von JMI, und zwar dann, wenn sich Veränderungen für die Generierung der Schnittstellen ergeben. Von Bedeutung ist dieser Nachteil allerdings eher für Module, die auf Modelldaten über die Schnittstellen zugreifen, nicht für die Datenhaltung.

Container für Projektdaten

Alle Projektdaten werden in Dateien dauerhaft gespeichert. Für den Spezifikationstext wird die Speicherung in unstrukturierten Textdateien beibehalten. Für die Modelldaten wird XMI als Dateiformat verwendet.[8] Während TESSI 1.1 die Projektdaten als eine Anzahl unabhängiger Dateien abspeicherte, soll TESSI 2.0 seine Dateien „gruppieren“. Eine einfache Lösung, die durch Java unterstützt wird und auch von anderen CASE-Werkzeugen angewandt wird, ist die Speicherung aller Dateien in ein Zip-Archiv.

TESSI 2.0 benutzt für seine Projektdateien die Dateierweiterung „.tessi“. Eine „tessi“-Datei ist ein Zip-Archiv mit folgenden Einträgen:

- `project.properties`: Eine Textdatei mit dem Format der „properties“-Dateien von Java. Jede Zeile entspricht einem Eintrag. Jeder Eintrag hat die Form `key=value`. `key` ist dabei ein definierter Schlüssel und `value` der diesem Schlüssel zugewiesene Wert.
- `specificationtext.txt`: Eine Textdatei, die den Spezifikationstext enthält.
- `model.xmi`: Eine XMI-Datei mit den Modelldaten.

Für die Datei `project.properties` wurden folgende Schlüssel definiert:

- `tessi.file.version` Die Version des Dateiformates. Im Moment ist der Wert `1.0` zu benutzen. Tritt ein anderer Wert auf, wird das File nicht geladen.

  Dieser Wert soll es ermöglichen, in der Zukunft unterschiedliche Dateiformate auseinanderzuhalten.

- `model.uml.version` Die Version der UML, mit der die Modelldaten modelliert sind. Für TESSI 2.0 ist der Wert 1.3 zu benutzen. Wird ein anderer Wert gefunden, dann wird das File nicht geladen.

  Dieser Eintrag soll es später ermöglichen, unterschiedliche Versionen der UML zu verwenden.

[8] Die Speicherung des Repositoryinhaltes in Form von XMI-Dateien ist eine „eingebaute“ Eigenschaft von JMI und damit auch von MDR.

- `project.name` Der Name des gespeicherten Projektes.

Weiterhin enthaltene Schlüssel werden ignoriert.

Aufgrund seiner Eigenschaften kann ein „tessi“-File mit einem gängigen Zip-Packprogramm ausgepackt werden. Der Inhalt ist also nicht dem Zugriff des Nutzers entzogen. Der Nutzer soll jedoch auch aus TESSI 2.0 heraus Zugriff auf die Bestandteile haben. Dafür wird ein modulspezifisches Menü vorgesehen. Ein Projekt ist in nur einer Datei enthalten, der Transport wird damit einfacher und sicherer.

5.2 Modul Basis

Aufgabenbeschreibung

Das Basismodul bildet den Kern der Anwendung TESSI. Es folgt eine Beschreibung der Verantwortlichkeiten des Moduls.

Bereitstellung des Dienstes Modulhandhabung Der Dienst wurde im Abschnitt 5.1.2 beschrieben. Für die dort genannten Rahmenbedingungen sind Festlegungen zu treffen. Das Modul „Basis“ hat eine Implementierung für die Klasse `ModuleLoader` bereitzustellen.

Bereitstellung des Nachrichtendienstes Der Dienst wurde in Abschnitt 5.1.3 beschrieben. Das Modul „Basis“ hat für eine Implementierung der beiden Interfaces `CommandRegistry` und `Command` zu sorgen. Bereitzustellen sind weiterhin die Klassen `Event`, deren direkte Kindklassen (siehe Klassendiagramm in Abbildung 5.3 auf Seite 56) und alle Klassen für Standardereignisse (für die Liste der identifizierten Standardereignisse siehe Anhang A). Bereitzustellen sind die Klasse `Action` und ihre direkten Kindklassen (siehe Klassendiagramm in Abbildung 5.3 auf Seite 56). Weiterhin sind Aktionen für all die `ApplicationEvents` zu implementieren, für die das Modul „Basis“ eine von anderen Modulen unabhängige Standardreaktion liefern muß. Das betrifft nur das `ApplicationCloseRequestEvent`.

Das Basismodul ist verantwortlich für die Registrierung aller Standardereignisse, auch derer, die nicht vom Modul bearbeitet werden.

Bereitstellung der GUI-Schnittstelle Der Dienst wurde in den Abschnitten 5.1.4 und 5.1.5 beschrieben. Das Modul „Basis“ muß eine Implementierung für das Interface `UIRegistry` und die Klassen `MainMenu` und `MenuEntry` bereitstellen.

Verwaltung der Dienste Das Basismodul ist im Zusammenhang mit der Modulhandhabung verantwortlich dafür, daß die vier Dienste Modulhandhabung, Datenhaltung, Nachrichtensystem und GUI-Erweiterung für die anderen Module zugreifbar sind. Dazu ist der Import des Dienstes Datenhaltung sicherzustellen und eine Implementierung des Interfaces `ServiceDirectory` bereitzustellen. Es ist außerdem sicherzustellen, daß der Programmablauf kontrolliert abgebrochen wird, wenn die nötigen Dienste nicht vorhanden sind.

Start der Applikation Das Modul muß den Start der Applikation TESSI realisieren.

Konstruktion und Verwaltung der gemeinsamen Oberfläche Das Modul muß die Konstruktion der gemeinsamen Oberfläche realisieren und die Anzeige der graphischen Komponenten der Module gewährleisten.

Emittierte Ereignisse

Die im Folgenden aufgelisteten Ereignisse werden ausnahmslos durch Befehle im Menü der gemeinsamen Oberfläche ausgelöst.

`NewProjectRequestEvent`

`OpenProjectRequestEvent`

`SaveProjectRequestEvent`

`SaveProjectAsRequestEvent`

`ApplicationCloseRequestEvent`

`UndoRequestEvent`

`RedoRequestEvent`

`CopySelectionRequestEvent`

`CutSelectionRequestEvent`

`PasteRequestEvent`

`DeleteSelectionRequestEvent`

`SelectAllRequestEvent`

`FindRequestEvent`

`FindNextRequestEvent`

`FindPreviousRequestEvent`

`ReplaceSelectionRequestEvent`

`HighlightRequestEvent`

`UnhighlightRequestEvent`

`RefreshRequestEvent`

`ShowHelpRequestEvent`

Weiterhin können beliebige weitere Ereignisse durch Modulmenüs ausgelöst werden. Voraussetzung dafür ist, daß ein Modul das entsprechende Ereignis bereitstellt, registriert und ein Modulmenü anmeldet.

Bearbeitete Ereignisse

Das Basismodul stellt lediglich eine Aktion zur Behandlung des TESSI-Ereignisses `ApplicationCloseRequestEvent`s zur Verfügung. Diese Aktion führt zur sofortigen Beendigung des Programmablaufs.

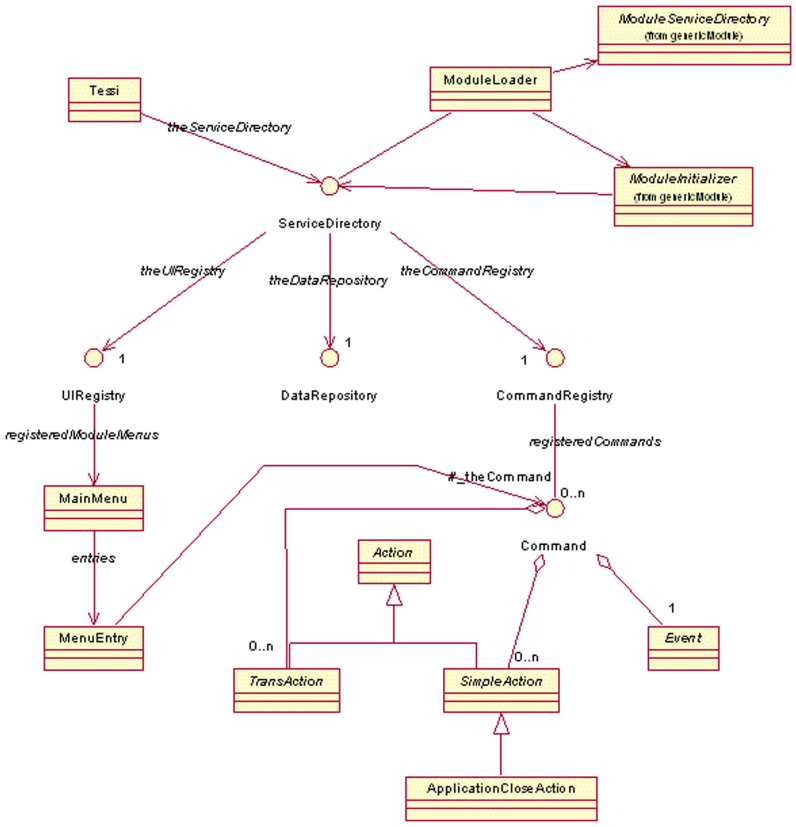

Abb. 5.6: Mögliches Modell für das Modul „Basis“.

5.3 Modul Datenhaltung

Aufgabenbeschreibung

Bereitstellung des Dienstes Datenhaltung Das Modul „Datenhaltung“ ist für die Bereitstellung des in Abschnitt 5.1.6 beschriebenen Dienstes verantwortlich. Neben der Implementierung muß auch der Export des Dienstes ermöglicht werden. Das Modul ist speziell für die Generierung von JMI-konformen Schnittstellen und einer Implementierung für diese Schnittstellen verantwortlich. Vorgesehen ist die Verwendung einer Fremdsoftware, die diese Aufgaben erfüllt: das Netbeans *Metadata Repository* (MDR). MDR übernimmt im Wesentlichen diese Aufgaben. Die „Datenhaltung“ ist jedoch für die Einbindung von MDR verantwortlich (Initialisierung, ggf. Anpassungen von Schnittstellen usw.).

Bereitstellung von Aktionen für Import und Export der Modelldaten Das Modul „Datenhaltung“ ist verantwortlich für die Bereitstellung von Aktionen, die die Speicherung und das Wiedereinlesen von Modelldaten durchführen (siehe bearbeitete Ereignisse).

Benachrichtigung über Veränderungen der Projektdaten Das Modul „Datenhaltung“ muß Veränderungen an Projektdaten über das Nachrichtensystem bekannt machen. Zur Vereinfachung für alle übrigen Module ist es notwendig, die spezifischen Nachrichten, die das MDR-Repository bereitstellt, in TESSI-Nachrichten zu übersetzen.

Implementierung des Containers für Projektdaten Der unter 5.1.6 beschriebene Container für die Projektdaten ist zu implementieren.

Emittierte Ereignisse

`SpecificationTextChangedEvent`

`RepositoryChangedEvent`

`ModelElementChangedEvent` (alle Ereignisse dieser Klasse)

Bearbeitete Ereignisse

`ApplicationCloseRequestEvent`	
Reaktion:	Wenn die aktuellen Projektdaten seit dem letzten Speichern modifiziert wurden, muß das Ereignis blockiert werden. Der Nutzer kann jedoch bestätigen, daß er TESSI ohne Speicherung beenden will, oder er speichert die Daten vor dem Beenden ab.
`NewProjectRequestEvent`	
Reaktion:	Anlegen eines neuen Projektes, d. h. alle aktuellen Daten werden verworfen und ein leeres Projekt wird angelegt. Im Falle ungespeicherter aktueller Daten gelten die gleichen Bedingungen wie bei `ApplicationCloseRequestEvent`.
`OpenProjectRequestEvent`	
Reaktion:	Laden von Projektdaten, d. h. alle aktuellen Daten werden verworfen. Aus dem Inhalt einer „tessi“-Datei wird ein neues Projekt angelegt. Im Falle ungespeicherter aktueller Daten gelten die gleichen Bedingungen wie bei `ApplicationCloseRequestEvent`.
`SaveProjectAsRequestEvent`	
Reaktion:	Alle aktuellen Daten werden in eine „tessi-Datei“ gespeichert. Der Name und Pfad der Datei müssen vorher vom Nutzer erfragt werden.
`SaveProjectRequestEvent`	
Reaktion:	Alle aktuellen Daten werden in eine „tessi“-Datei gespeichert. Der Name und Pfad der Datei müssen aus einem früheren Lade- oder Speichervorgang bekannt sein. Sind diese Informationen nicht bekannt, ist die Reaktion wie bei `SaveProjectAsRequestEvent`.

`ExportModelToXMIRequestEvent`	
Reaktion:	Die aktuellen Modelldaten werden in eine XMI-Datei gespeichert. Name und Pfad der XMI-Datei sind vorher vom Nutzer zu erfragen. Der Export gilt nicht als Speicherung, d. h. modifizierte Daten gelten nach dem Export immer noch als modifiziert.

`ImportXMIModelRequestEvent`	
Reaktion:	Die aktuellen Modelldaten werden verworfen und neue Modelldaten werden aus einer XMI-Datei geladen. Name und Pfad der XMI-Datei sind vorher beim Nutzer zu erfragen. Der Nutzer ist vor dem Vorgang zu warnen. Im Falle ungespeicherter aktueller Daten gelten die gleichen Bedingungen wie bei `ApplicationCloseRequestEvent`.

`ExportSpecificationTextRequestEvent`	
Reaktion:	Der aktuelle Spezifikationstext wird als Textdatei gespeichert. Name und Pfad der Textdatei sind vorher vom Nutzer zu erfragen. Der Export gilt nicht als Speicherung, d. h. modifizierte Daten gelten nach dem Export immer noch als modifiziert.

`ImportSpecificationTextRequestEvent`	
Reaktion:	Der aktuelle Spezifikationstext wird verworfen und ein neuer Spezifikationstext wird aus einer Textdatei geladen. Name und Pfad der Textdatei sind vorher beim Nutzer zu erfragen. Der Nutzer ist vor dem Vorgang zu warnen. Im Falle ungespeicherter aktueller Daten gelten die gleichen Bedingungen wie bei `ApplicationCloseRequestEvent`.

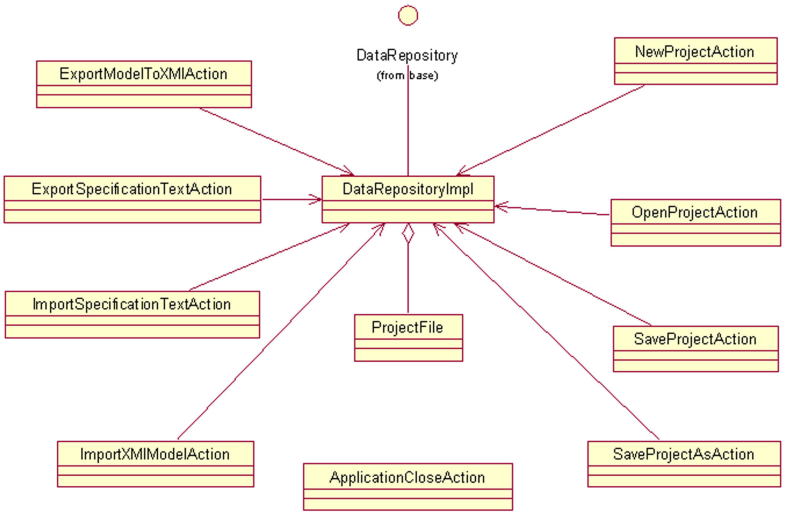

Abb. 5.7: Mögliches Modell für das Modul „Datenhaltung“.

5.4 Modul Spezifikationstext

Aufgabenbeschreibung

Editieren eines Spezifikationstextes Der Nutzer benutzt den bereitgestellten Editor, um einen Spezifikationstext zu erarbeiten bzw. bearbeiten. Es sind die typischen, die Textbearbeitung unterstützenden Handlungen möglich, wie z. B. Suchen, Ersetzen, Kopieren, Ausschneiden, Einfügen usw. Als Besonderheit von TESSI existiert noch eine Funktion zum Hervorheben von Suchbegriffen im gesamten Text.

Hervorheben von Textelementen Namen und Alia von einigen Modellelementen werden im Text farbig hervorgehoben. Welche Modellelemente dafür in Frage kommen, ist durch TESSI 1.1 festgelegt. Es sind dies: Klassen, Methoden, Attribute und Zustände.

Bereitstellung von Steuerungselementen Dem Nutzer werden Steuerungselemente bereitgestellt, die ihm ermöglichen, typische Handlungsabläufe durchzuführen. Es soll möglich sein, ausgehend von markierten Textstellen Modellelemente anzulegen. Über die Tatsache, daß Text markiert wurde und welcher Text das ist, sollen TESSI-Nachrichten erzeugt werden.

Emittierte Ereignisse

`TextSelectionByUserEvent`

`CreateModelElementRequestEvent` (alle Ereignisse dieser Klasse)

`ShowHelpRequestEvent`

Bearbeitete Ereignisse

`SpecificationTextChangedEvent`	
Reaktion:	Der aktuelle Text im Editor wird verworfen und der neue Spezifikationstext wird aus dem `DataRepository` geladen. Danach werden die farbigen Hervorhebungen im Text neu aufgebaut.

`NewProjectRequestEvent`	
Reaktion:	Der aktuelle Text im Editor wird in das `DataRepository` zurückgeschrieben.

`SaveProjectAsRequestEvent`	
Reaktion:	Der aktuelle Text im Editor wird in das `DataRepository` zurückgeschrieben.

`SaveProjectRequestEvent`	
Reaktion:	Der aktuelle Text im Editor wird in das `DataRepository` zurückgeschrieben.

`CopySelectionRequestEvent`	
Reaktion:	Die aktuelle Markierung im Text wird in die Zwischenablage kopiert.

`CutSelectionRequestEvent`	
Reaktion:	Die aktuelle Markierung im Text wird in die Zwischenablage kopiert. Der markierte Textbereich wird anschließend gelöscht.

`DeleteSelectionRequestEvent`	
Reaktion:	Der markierte Textbereich wird gelöscht.

`PasteRequestEvent`	
Reaktion:	Der aktuelle Inhalt der Zwischenablage wird an der aktuellen Caretposition in den Text eingefügt.

`RefreshRequestEvent`	
Reaktion:	Baut die farbige Markierung von Textteilen neu auf.

`SelectAllRequestEvent`	
Reaktion:	Markiert den gesamten aktuellen Text.

`FindNextRequestEvent`	
Reaktion:	Sucht das nächste Vorkommen des letzten, in den Suchdialog eingegebenen Suchbegriffes. Keine Reaktion, wenn kein solcher Begriff existiert. Neben dem Suchbegriff werden auch die zuletzt verwandten Suchoptionen benutzt (Groß-/Kleinschreibung, nur ganze Wörter).

`FindPreviousRequestEvent`	
Reaktion:	Sucht das vorige Vorkommen des letzten, in den Suchdialog eingegebenen Suchbegriffes. Keine Reaktion, wenn kein solcher Begriff existiert. Neben dem Suchbegriff werden auch die zuletzt verwandten Suchoptionen benutzt (Groß-/Kleinschreibung, nur ganze Wörter).

`FindRequestEvent`	
Reaktion:	Öffnet den Suchen-Dialog

`HighlightRequestEvent`	
Reaktion:	Einfache Hervorhebung aller Vorkommen des zuletzt gesuchten Suchbegriffes im aktuellen Text.

`UnhighlightRequestEvent`	
Reaktion:	Macht die Hervorhebung rückgängig (nur die Hervorhebungen durch `HighlightRequestEvent`).

`ModelElementChangedEvent`	
Reaktion:	Je nach konkreter Benachrichtigung werden Name und Alia eines Modellelementes hervorgehoben (Modellelement wurde erzeugt), die Hervorhebungen für das Modellelement zurückgesetzt (Modellelement wurde gelöscht) oder auf andere Textteile verschoben (Name oder Alia haben sich geändert)
Hinweis:	Es werden momentan Veränderungen an nur vier Modellelementen ausgewertet. Die Ereignisklassen dazu lauten: `ClassChangedEvent` für Klassen, `AttributeChangedEvent` für Attribute, `MethodChangedEvent` für Methoden und `StateChangedEvent` für Zustände. Für andere Modellelemente gibt es keine farbigen Markierungen.

Graphische Komponenten

Zur Funktion des Moduls werden ein Editor, ein „Suchen“- und ein „Ersetzen“-Dialog benötigt. Für den Editor kann die in TESSI 1.1 verwendete Komponente wiederverwendet werden (`Tessi.ExtTextPane`). Gleiches gilt für die im Editor verwendete Dokumentenklasse (`Tessi.UndoableStyledDocument`). Es sind jedoch möglicherweise Anpassungen nötig. Der Editor ist im Bereich `EDITOR_POSITION` anzuzeigen.

In TESSI 1.1 kommt ein kombinierter Suchen- und Ersetzendialog zum Einsatz. Dieser sollte wiederverwendet werden.

Für die Bereitstellung von Steuerelementen wird ein Kontextmenü benötigt. Dazu können Standardklassen der Swing-Bibliothek benutzt werden.

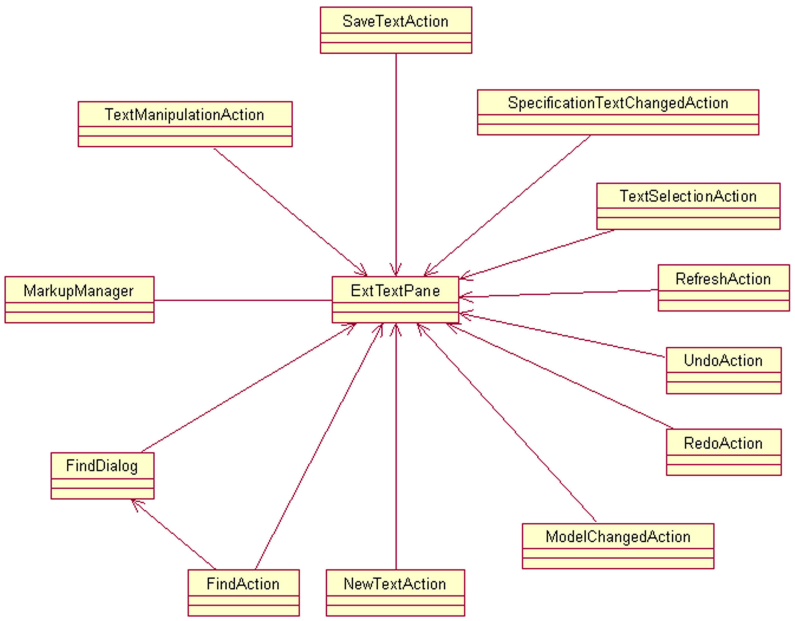

Abb. 5.8: Mögliches Modell für das Modul „Spezifikationstext“

5.5 Modul Modellübersicht

Aufgabenbeschreibung

Darstellung der Modelldaten als Baum Ein Teil der vorhandenen Modelldaten wird als Baum dargestellt. Es soll deutlich erkennbar sein, von welchem Typ ein angezeigtes Modellelement ist und mit welchen anderen Modelldaten es in Beziehung steht.

Bereitstellung von Steuerungselementen Für alle dargestellten Modellelemente sollen Steuerungselemente verfügbar sein, die die Bearbeitung des Modellelementes, das Löschen des Modellelementes und die Erzeugung von Modellelementen des gleichen Typs veranlassen.

Benachrichtigung über Auswahl von Modellelementen Über die Tatsache, daß ein Modellelement ausgewählt wurde und welches das ist, sollen TESSI-Nachrichten generiert werden.

Emittierte Ereignisse

`CreateModelElementRequestEvent` (alle Ereignisse dieser Klasse)

`DeleteModelElementRequestEvent`

`EditModelElementRequestEvent`

`ModelElementSelectionByUserEvent`

`ShowHelpRequestEvent`

Bearbeitete Ereignisse

`ModelElementChangedEvent`	
Reaktion:	Je nach konkreter Benachrichtigung werden Modellelemente eingeblendet (Modellelement wurde erzeugt), ausgeblendet (Modellelement wurde gelöscht), verschoben (Modellelement wurde einem anderen Modellelement zugeordnet) oder die Darstellung geändert (Name hat sich geändert)
`RepositoryChangedEvent`	
Reaktion:	Die aktuelle Darstellung wird verworfen und eine neue Darstellung wird anhand des geänderten Repositoryinhaltes aufgebaut.
`RefreshRequestEvent`	
Reaktion:	Wie `RepositoryChangedEvent`. Hier ist nur ein anderes Ereignis der Auslöser.

Graphische Komponenten

Das Modul benötigt eine graphische Komponente, die die Baumansicht enthält. Sie ist im Bereich `BIG_VIEW_POSITION` anzuzeigen. Darüber hinaus werden Kontextmenüs benötigt.

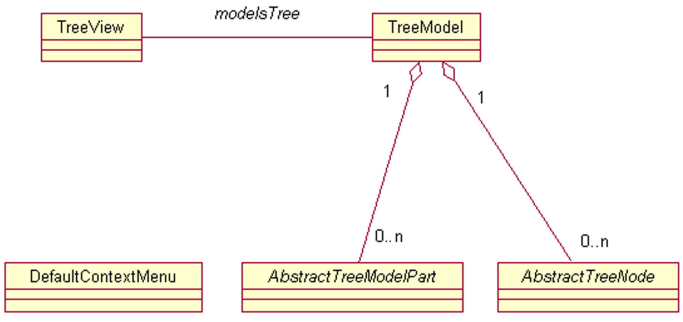

Abb. 5.9: Mögliches Modell für das Modul „Modellübersicht“

5.6 Modul Modellmanipulation

Aufgabenbeschreibung

Bereitstellung von Dialogen zur Modelldateneingabe Für alle UML-Modellelemente, die von TESSI 2.0 zur Modellierung benutzt werden, sollen Dialoge zur Verfügung stehen, die die Erzeugung, das Löschen und die Bearbeitung von Modellelementen ermöglichen.

Bereitstellung von Dialogen zur Modellexpansion Die aus TESSI 1.1 bekannten Expansionsdialoge sind zur Verfügung zu stellen.

Emittierte Ereignisse

`ModelElementSelectionByUserEvent`

`ShowHelpRequestEvent`

Bearbeitete Ereignisse

`CreateModelElementRequestEvent` (alle Unterklassen)	
Reaktion:	Der Dialog zur Bearbeitung des betreffenden Modellelementetyps wird geöffnet und mit Standardwerten vorbelegt. Der Name des neuen Modellelementes wird auf den zuletzt markierten Text gesetzt.
`ModelElementChangedEvent` (alle Unterklassen)	
Reaktion:	Alle aktuell offenen Dialoge laden die geänderten Daten neu aus dem `DataRepository`
`EditModelElementRequestEvent`	
Reaktion:	Der Dialog zur Bearbeitung des betreffenden Modellelementetyps wird geöffnet und mit den Daten des aktuell markierten Modellelementes geladen.
`DeleteModelElementRequestEvent`	
Reaktion:	Das aktuell markierte Modellelement wird gelöscht.
`RepositoryChangedEvent`	
Reaktion:	Die Daten aller aktuell offenen Dialoge werden verworfen und neue Modelldaten werden aus dem `DataRepository` geladen.
`RefreshRequestEvent`	
Reaktion:	Wie `RepositoryChangedEvent`. Hier ist nur ein anderes Ereignis der Auslöser.
`ModelElementSelectionByUserEvent`	
Reaktion:	Die Information, welches Modellelement aktuell markiert ist, wird zur späteren Verwendung gespeichert.
`TextSelectionByUserEvent`	
Reaktion:	Der aktuell markierte Text wird zur späteren Verwendung gespeichert.

Graphische Komponenten

Das Modul stellt Dialoge bereit, die nicht als graphische Komponenten im hier verwendeten Sinn gelten.

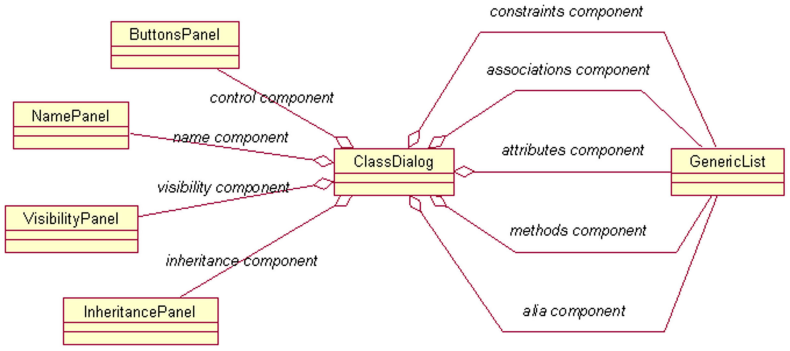

Abb. 5.10: Teil eines möglichen Modells zum Modul „Modellmanipulation“ (Eigenschaftsdialog zum Modellelement `Class`)

5.7 Modul Thesaurus

Aufgabenbeschreibung

Einbindung des Thesaurus WordNet Das Modul muß dabei auf Textmarkierung reagieren und entsprechend vorbereitete Suchoptionen für den Thesaurus bereitstellen. Die Suchergebnisse werden analog zu TESSI 1.1 in einer eigenen Textansicht dargestellt.

Emittierte Ereignisse

`ShowHelpRequestEvent`

Bearbeitete Ereignisse

`TextSelectionByUserEvent`	
Reaktion:	Der markierte Text wird übernommen und es werden darauf abgestimmte Suchoptionen vorformuliert.

Graphische Komponenten

Das Modul muß eine Eingabemaske bereitstellen, die alle möglichen Suchoptionen gruppiert. Der Nutzer soll auch die Möglichkeit haben, einen Suchbegriff in der Maske einzugeben. Als Position in der gemeinsamen Oberfläche empfiehlt sich `SMALL_VIEW_POSITION`. Die Ausgabe erfolgt, wie in TESSI 1.1 in einer nicht editierbaren Textansicht (Oberflächenbereich `EDITOR_POSITION`). Wenn für eine Suche das Ergebnis vorliegt, muß das Anzeigefenster in den Vordergrund geschaltet werden.

5.8 Modul Textgenerierung

Aufgabenbeschreibung

Generierung von Texten Aus den vorhandenen Modelldaten werden Texte verschiedenen Formates generiert. Die Ausgabe der Texte erfolgt wie in TESSI 1.1 in nicht editierbaren Textansichten. In TESSI 2.0 werden nicht alle Textformate aus TESSI 1.1 aufgenommen. Übernommen werden „*static text*“, „*dynamic text*“ und der Text der Metrikfunktion. Die Erzeugung von Quelltexten entfällt.

Emittierte Ereignisse

`ShowHelpRequestEvent`

Bearbeitete Ereignisse

`RefreshRequestEvent`	
Reaktion:	Alle aktuellen generierten Texte werden verworfen und aus den aktuellen Modelldaten neu erzeugt.

`RepositoryChangedEvent`	
Reaktion:	Wie bei `RefreshRequestEvent`. Hier haben sich allerdings alle Modelldaten geändert.

`ExportGeneratedTextsRequestEvent`	
Reaktion:	Die generierten Texte werden in Textdateien geschrieben. Der Name und Pfad der Textdateien ist vorher vom Nutzer zu erfragen.
Hinweis:	Dieses Ereignis ist modulspezifisch und sollte in ein Modulmenü aufgenommen werden.

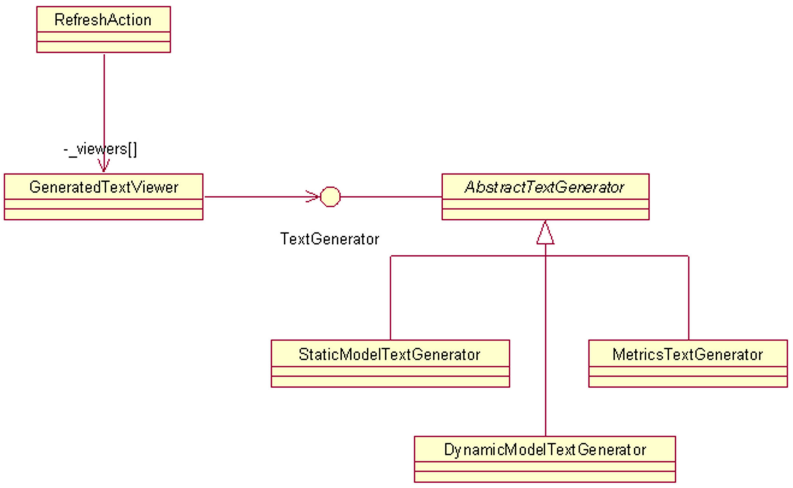

Abb. 5.11: Mögliches Modell für das Modul „Textgenerierung“

5.9 Modul Hilfe

Aufgabenbeschreibung

Anzeige von Hilfe Der Nutzer kann durch Betätigung von Steuerelementen Hilfe anfordern. Während die Hilfetexte in Form von HTML-Dateien durch die Module selbst zu erbringen sind, bietet das Modul Hilfe ein Mittel für die anderen Module, mit dem sie Hilfetexte anzeigen lassen können.

Emittierte Ereignisse

`ShowHelpRequestEvent`

Bearbeitete Ereignisse

`ShowHelpRequestEvent`	
Reaktion:	Der als Parameter im Ereignis übermittelte Hilfeidentifikator ist auszuwerten und der dazu gehörende Hilfetext ist anzuzeigen.

5.10 Adapter-Bibliothek

Der Zugriff auf die Modelldaten erfolgt über Interfaces, die gemäß der JMI-Spezifikation aus dem UML-Metamodell generiert werden. Dabei werden die Struktur der UML sowie die Gesamtheit aller im Metamodell vorhandenen Modellelemente und deren Eigenschaften in Java-Konstrukte (Interfaces, Methoden) überführt. Die erzeugten Interfaces und ihre Beziehungen weisen also eine der UML entsprechende Komplexität auf.

Bei der Programmierung auf diese JMI-Schnittstelle hin wird schnell klar, daß bereits einige relativ einfache Aufgaben eine große Menge an Code erfordern. Das liegt zum einen an der notwendigen Navigation in der Package-Struktur der UML. Andererseits müssen für TESSI Aufgaben erfüllt werden, die, in die Sprache der UML übersetzt, eine größere Anzahl granularer Einzelhandlungen erfordern. Schließlich gibt es auch Aufgaben, die häufig benötigt werden. Für alle diese Fälle wurde eine Adapter-Bibliothek entworfen.

Für jedes von TESSI aus dem Umfang der UML verwendete Modellelement wurde ein Interface definiert, welches Methoden für Aufgaben in TESSI bereitstellt. Dazu kommen noch Interfaces für Klassenproxys von benutzten Modellelementen. Hier ist in der Regel die Erzeugung von Instanzen definiert.[9] Als der primäre Zugriffspunkt für die Bibliothek dient das Interface `UmlAdapterFactory`. Es definiert Methoden für den Zugriff auf die Klassenproxys der verwendeten Modellelemente. Die Navigation in den Paketstrukturen der UML entfällt damit.

Zum Namen „Adapter“ ist schließlich noch anzumerken: Ein Adapter paßt laut Literatur die Schnittstelle einer Klasse an eine andere an.[10] Das ist hier zweifellos der Fall und deshalb wurde der Name gewählt.

[9] Einige Modellelemente der UML werden benutzt, ohne daß sie explizit in TESSI zum Vorschein kommen. Ein Beispiel ist `TaggedValue`. Mit Hilfe dieses Modellelementes werden Dokumentation, Alia und andere Eigenschaften gespeichert.

[10] [GoF96, S. 171] oder [Oes01, S. 79]

6. STANDARDIMPLEMENTIERUNG TESSI 2.0

Nach der Spezifizierung der Komponenten von TESSI 2.0 geht es nun um Eckpunkte der Implementierung. In diesem Kapitel wird beschrieben, welche Mittel und Techniken verwendet wurden, um die in Kapitel 5 formulierten Anforderungen zu erfüllen. Die beiden Kapitel stehen damit in engem Zusammenhang.

Die Abschnitte zur Implementierung der Module sollen einen Einstieg in das Verständnis geben. Die konkrete Implementierung ist detaillierter in der Quellcodedokumentation beschrieben. Zu weiteren Informationen wird deshalb auf die API-Dokumentation verwiesen.

6.1 Allgemeine Festlegungen

In diesem Abschnitt werden einige Details der technischen Umsetzung beschrieben, die zum Verständnis der Implementierung wichtig sind.

6.1.1 Build-Umgebung

Unter dem Stichwort *Build* wird die Erzeugung eines ablauffähigen Programms aus dem Quelltext sowie aus anderen Beschreibungen verstanden. Mit *Build*-Umgebung ist dann die Gesamtheit aller Werkzeuge und Mechanismen gemeint, die dies durchführen und steuern.

Als Werkzeug zum Steuern des *Build*-Prozesses von TESSI kommt das freie Werkzeug Ant[1] zum Einsatz. Ant basiert auf Java. Es bringt eine Menge von Fähigkeiten mit, die speziell auf das Bauen von Java-basierter Software ausgerichtet sind. Es ist außerdem erweiterbar. Viele wichtige Werkzeuge für Java und auch einige der schon erwähnten *Reengineering*-Werkzeuge, wie JavaNCSS, lassen sich in Ant integrieren.

Um Ant einsetzen zu können, ist der *Build*-Vorgang zunächst anhand von Regeln zu beschreiben.[2] Das für TESSI 2.0 gültige File `build.xml` befindet sich im Stammverzeichnis der Quellendistribution (siehe nächster Abschnitt). Es sind folgende *Build*-Ziele vorhanden:

- `all`: Erzeugt das ablauffähige Programm TESSI 2.0 aus den Quellen. Die nötige Verzeichnisstruktur und Startskripte für Windows und Linux werden mit erzeugt (siehe Abschnitt 6.1.2). `all` ist das Standardziel für *Builds*.
- `interfaces`: Erzeugt JMI-Interfaces aus der XMI-Beschreibung der UML. Dieser Vorgang muß normalerweise nicht mehr durchgeführt werden.

[1] http://ant.apache.org/

[2] Zur Verwendung von Ant und zum Format der Regeldatei sei auf die Hilfe zu Ant verwiesen.

- `dist`: Synonym für `all`; siehe dort.
- `metrics`: Erzeugt eine JavaNCSS-Analyse des zu TESSI 2.0 gehörenden Quelltextes (als HTML-Datei).

Die typischerweise vorhandenen Subtasks `compile`, `javadoc` usw. sind ebenfalls definiert. Allerdings erzeugt nur `dist` (bzw. `all`) ein lauffähiges TESSI. Mit dem üblichen Task `clean` können alle im *Build* erzeugten Dateien gelöscht werden.

Alternativ kann TESSI 2.0 auch aus der IDE Eclipse heraus gebaut werden. Dazu ist die Ant-Unterstützung von Eclipse zu nutzen.

6.1.2 Verzeichnisstruktur

Für TESSI 2.0 wurde eine Verzeichnisstruktur festgelegt, in der die benötigten Komponenten abgelegt werden. Abbildung 6.1 zeigt die komplette Struktur der *Build*-Umgebung von TESSI 2.0.

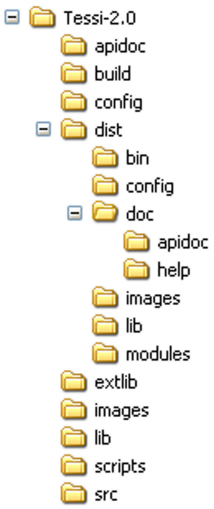

Abb. 6.1: Struktur der Verzeichnisse in TESSI 2.0 (komplette *Build*-Umgebung)

Das im Bild gezeigte Verzeichnis `TESSI-2.0` ist das Stammverzeichnis der Quellendistribution von TESSI 2.0. Darin befinden sich das *Build*-Skript für Ant und Eclipse-Projektdateien. Die verschiedenen Unterverzeichnisse enthalten Bibliotheken, Skripte und Konfigurationsdaten, die für die Funktion von TESSI oder den *Build*-Vorgang benötigt werden.

Das Unterverzeichnis `TESSI-2.0/dist` ist das Basisverzeichnis von TESSI 2.0 als ausführbarem Programm. Es wird während des *Build*-Vorganges mit Ant erzeugt. Der Name des absoluten Pfads dieses Verzeichnisses wird dabei in die generierten Startskripte eingetragen. Beim Start von TESSI 2.0 wird die Angabe als Java-Systemproperty `tessi.base.dir` gesetzt. Anhand des Wertes dieser *Property* wird die Position wichtiger Unterverzeichnisse abgeleitet, wie das Moduleverzeichnis (`$tessi.base.dir/modules`), das Konfigurationsverzeichnis (`$tessi.base.dir/config`) sowie das Verzeichnis für API-Dokumentation und Hilfetexte (`$tessi.base.dir/doc`).

Nach einem *Build* von TESSI kann der ausführbare Teil (Verzeichnis `dist` und alles darunter) aus der *Build*-Umgebung an einen anderen Ort verschoben werden. Der Name des `dist`-Verzeichnisses kann ebenfalls verändert werden. Es sind jedoch die Startskripte anzupassen. Wird der Wert der *Property* falsch gesetzt oder kann auf die gesuchten Unterverzeichnisse aus irgendeinem Grund nicht zugegriffen werden, wird der Startvorgang von TESSI abgebrochen.

6.2 Modulhandhabung

Module für TESSI 2.0 werden im Verzeichnis `$tessi.base.dir/modules` erwartet. Sämtliche Ressourcen eines Moduls sind in der Regel in einer JAR-Datei[3] unterzubringen. Ausnahmen von dieser Regel sind in den Abschnitten zu den Modulen dokumentiert.

Jedes Modul muß einen `ModuleInitializer` enthalten, d. h. eine Implementierung für das Interface `de.tuc.tessi.base.ModuleInitializer`. Diese Klasse ist verantwortlich für die Initialisierung des Moduls. Ein Modul kann eine Klasse enthalten, die als Verzeichnis von exportierten Diensten des Moduls dient. Sie muß das Interface `de.tuc.tessi.base.ModuleServiceDirectory` implementieren. In TESSI 2.0 sollte nur das Modul eine solche Klasse enthalten, welches die Datenhaltung implementiert.[4]

Für das Manifest eines JAR-Files sind die folgenden beiden Schlüssel definiert:

- `ModuleInitializer`: Name der Klasse im Modul, welche das Interface `de.tuc.tessi.base.ModuleInitializer` implementiert
- `ModuleServiceDirectory`: (optional) Name der Klasse im Modul, welche das Interface `de.tuc.tessi.base.ModuleServiceDirectory` implementiert

Beim Laden des Moduls wird das Manifest eines JAR-Files nach diesen beiden Einträgen durchsucht. Enthält das Manifest keinen der beiden Einträge, wird das JAR-File als ungültig verworfen.

In den anderen Fällen wird versucht, die angegebenen Klassen zum Manifest-Eintrag `ModuleServiceDirectory` zu instantiieren. In TESSI 2.0 definiert das Interface `de.tuc.tessi.base.ModuleServiceDirectory` nur eine Methode, die eine Implementierung des Interfaces `de.tuc.tessi.base.DataRepository` liefert. Diese Methode wird benutzt, um die Datenhaltung im `ServiceDirectory` für alle anderen Module verfügbar zu machen. Es wird genau eine Implementierung eines `DataRepository` zugelassen. Alle anderen Fälle führen zum Programmabbruch.

Schließlich wird für jedes Modul versucht, die angegebene Klasse für den Manifest-Eintrag `ModuleInitializer` zu instantiieren. Durch Aufruf der Methode `initializeModule` für jeden `ModuleInitializer` werden die Module aktiviert. Beim Aufruf der Methode wird das `ServiceDirectory` als Parameter übergeben, wodurch dem Modul die importierten Dienste bekannt gemacht werden.

Zusätzlich zu den angegebenen Abbruchbedingungen führt jeder beliebige Fehlerzustand beim Laden der Module zum Programmabbruch. Dies ist nötig, da die Wichtigkeit der Module nicht ausgewertet wird und so nicht entschieden werden kann, ob ein nicht ladbares Modul verzichtbar ist.

Es wird außerdem davon ausgegangen, daß Module wissentlich und willentlich in das Moduleverzeichnis bewegt werden. Läßt sich ein Modul-JAR nicht laden, so stimmt entweder diese Annahme nicht, oder das Modul ist insgesamt fehlerhaft. In jedem Fall sollte der Fehlerzustand zunächst beseitigt werden.

[3] Informationen zum Dateiformat JAR und zum erzeugenden Programm `jar` sind in der Hilfe zu jedem aktuellen *Java Development Kit* (JDK) zu finden.

[4] Zur Programmierung eines Moduls siehe Anhang C.

6.3 Nachrichtensystem

Das Nachrichtensystem ist gemäß dem Klassendiagramm aus Abbildung 5.3 umgesetzt worden. Aus dem Diagramm gehen einige Details nicht hervor, die hier verdeutlicht werden.

Der Aufruf der beiden Arten von `Action`s als Reaktion auf eingetretene Ereignisse geschieht wie in Abschnitt 5.1.3 beschrieben. Insgesamt sind dabei die folgenden Schritte abzuarbeiten:

1. Aufruf der `prepare`-Methoden aller `TransAction`s, die für die Ereignisklasse oder für eine ihrer Oberklassen registriert sind.

2. (a) (wenn in Schritt 1 mindestens ein Veto aufgetreten ist) Aufruf der `cancel`-Methoden aller `TransAction`s, die für die Ereignisklasse oder für eine ihrer Oberklassen registriert sind.

   (b) (wenn in Schritt 1 kein Veto aufgetreten ist) Aufruf der `commit`-Methoden aller `TransAction`s und anschließend Aufruf der `perform`-Methoden aller `SimpleAction`s, die für die Ereignisklasse oder für eine ihrer Oberklassen registriert sind.

Für jeden dieser Schritte ist in der Klasse `CommandImpl`, der Implementierung des Interfaces `Command`, eine Methode vorgesehen. Die Methoden bearbeiten alle `Action`s, welche für die Ereignisklasse des `Command`s registriert wurden. Den Aufruf dieser Methoden für alle `Command`s aller Ereignisklassen (ausgehend vom eingetretenen Ereignis die Kette der Superklassen hinauf bis einschließlich `Event`) koordiniert die Methode `processEvent` der Klasse `CommandRegistryImpl` (die Implementierung von `CommandRegistry`). Instanzen von `CommandImpl` delegieren die erhaltenen Ereignisse zur Bearbeitung an diese Methode.

Die beschriebene Kollaboration zwischen den Klassen ist nötig, um einerseits die Bearbeitung in den `Command`s der Oberklassen des Ereignisses auszulösen. Andererseits muß für `TransAction`s, die für Oberklassen des Ereignisses registriert sind, ebenfalls das Auftreten von Vetos geprüft werden. Da die Verwaltung der vorhandenen `Command`s in die Verantwortung der `CommandRegistry` fällt, kann hier leicht nach eventuellen `Command`s für Oberklassen des aufgetretenen Ereignisses gesucht werden.[5] Existieren solche `Command`s, wird ihnen das Ereignis zur Bearbeitung delegiert.

Der Aufruf der Bearbeitungsmethoden der registrierten `Action`s ist durch *Catch-All*-Blöcke[6] geschirmt. Diese sollen verhindern, daß fehlerhafte Implementierungen von Aktionen die Bearbeitung durcheinander bringen. Außerdem werden so die Aufruffolgen für `TransAction`s garantiert. Das sind entweder (`prepare`, `commit`) oder (`prepare`, `cancel`).

[5] Zur Suche nach Oberklassen werden die reflektiven Fähigkeiten von Java verwendet. Der implementierte Mechanismus ist nicht anfällig für Veränderungen in der Vererbungsstruktur der Ereignisklassen.

[6] `try-catch`-Blöcke, die alle möglichen Ausnahmen abfangen. In Java kann z. B. die Superklasse der meisten Ausnahmen `java.lang.Exception` gefangen werden. Das Fangen von `java.lang.Error` sollte dagegen unterbleiben, da diese Ausnahme „katastrophales Versagen“ anzeigt (siehe Javadoc zu dieser Klasse).

6.4 Graphische Komponenten und Modulmenüs

Die Implementierung des Interfaces `UIRegistry` (Klasse `UIRegistryImpl`) ist sehr einfach. Sie verwaltet lediglich die registrierten Komponenten. Die das Hauptfenster implementierende Klasse `MainFrame` fragt die Informationen ab und benutzt sie zur Darstellung. Die Methode `switchTo` delegiert den Aufruf an `MainFrame`.

Der Vorteil der einfachen Schnittstelle ist darin zu sehen, daß es kaum Einschränkungen gibt, wie die Forderungen aus Abschnitt 5.1.4 mit GUI-Mitteln umzusetzen sind. So wurde z. B. die Darstellung mit Karteireitern gewählt, um die Komponenten eines Bereiches anzuordnen. Dafür ließe sich aber auch eine Vielzahl anderer Lösungen finden. Nachteilig ist die beschränkte Funktionalität, die jedoch für die aktuellen Anforderungen ausreicht.

Die Umsetzung der Steuerungsmöglichkeiten für Module, wie im Abschnitt 5.1.5 beschrieben, beschränkt sich ebenfalls auf eine einfache Lösung. Registrierte Modulmenüs werden als Submenüs des Hauptmenüs *Module* dargestellt. Zukünftige Implementierungen könnten z. B. gleichzeitig entsprechende modulspezifische Werkzeugleisten einblenden.

Für Ereignisse, die über Modulmenüs ausgelöst werden, gilt noch eine Einschränkung: da bei Betätigung eines GUI-Steuerelements durch den Nutzer das zugeordnete Ereignis erzeugt wird, muß aus der Ereignisklasse über einen öffentlichen, parameterlosen Konstruktor eine Instanz erzeugbar sein. Menübefehle, deren `Event`s diese Bedingung nicht erfüllen, werden nicht in das Menü aufgenommen.

6.5 Modul Basis

Packages:	`de.tuc.tessi.base`
Moduldatei:	`base-1.0.jar`

Die Packages des Moduls befinden sich in der JAR-Datei `base-1.0.jar`. Als einzige Moduldatei ist sie im Unterverzeichnis `$tessi.base.dir/lib` und nicht im Moduleverzeichnis untergebracht.

Auf die Implementierung der Dienste wurde bereits in den Abschnitten 6.2 bis 6.4 eingegangen.

Der Start der Applikation wird von der Klasse `Tessi` gesteuert.

Die Klasse `ServiceDirectoryImpl` ist die geforderte Implementierung des Interfaces `ServiceDirectory`. Der Start aller verwalteten Dienste ist in dieser Klasse konzentriert (außer der Datenhaltung, die vom gleichnamigen Modul importiert wird). Neben den bekannten vier Diensten wird auch die Konstruktion der Oberfläche (Klasse `MainFrame`) und die Registrierung der Standardnachrichten (Klasse `ApplicationInitializer`) von `ServiceDirectoryImpl` gesteuert.

6.6 Modul Datenhaltung

Packages:	`de.tuc.tessi.dataRepository`
Moduldatei:	`dataRepository-1.0.jar`

Zum Modul gehören neben der Moduldatei weiterhin sechs JAR-Dateien, welche das MDR enthalten bzw. dafür benötigt werden. Sie wurden nicht mit

in das JAR-File des Moduls aufgenommen, sondern sind einzeln im Verzeichnis `$tessi.base.dir/lib` enthalten. Ebenfalls in diesem Verzeichnis befindet sich eine Datei `uml.jar`, die die generierten Interfaces zur UML 1.3 enthält. Sie kann durch Aufruf des *Buildtargets* `interfaces` aus der XMI-Beschreibung der UML erzeugt werden. Die dafür benötigten XMI-Dateien (`01-12-02.xml` und `01-12-02_Diff.xml`)[7] werden auch zur Laufzeit gebraucht und befinden sich in den `config`-Unterverzeichnissen von *Build-* und Programmumgebung.

Die von Netbeans MDR bereitgestellte Funktionalität wird von der Klasse `DataRepositoryImpl` gekapselt. Sie ist auch die Implementierung für das Interface `DataRepository`. Die Reaktionen auf die im Abschnitt 5.3 genannten, zu bearbeitenden Ereignisse werden durch Klassen realisiert, deren Name jeweils das Suffix `Action` besitzt.

Das im Abschnitt 5.1.6 beschriebene Dateiformat für das Projektfile wird durch die Klasse `ProjectFile` implementiert. Die Klasse `DataRepositoryImpl` aggregiert jeweils eine Instanz von `ProjectFile` und transferiert die Daten über entsprechende Methoden.

Bei der Erarbeitung eines Konzeptes zur Überwachung von Veränderungen im Repository wurden einige Probleme festgestellt. Diese betrafen vor allem die Interpretation der MDR-spezifischen Benachrichtigungen. Parallel dazu war bereits die in Abschnitt 5.10 beschriebene Bibliothek von Adapterklassen im Aufbau. Es wurde entschieden, die Benachrichtigungen über Veränderungen im Repository an diese Bibliothek anzulehnen und die Ereignisse entsprechend zu definieren. Klienten ist jedoch immer auch der Zugriff auf die „reinen“ JMI-Konstrukte möglich.

6.7 Modul Spezifikationstext

Packages:	`de.tuc.tessi.specificationText`
Moduldatei:	`specificationText-1.0.jar`

Die graphischen Komponenten für Editor und Suchen-/Ersetzendialog konnten aus TESSI 1.1 übernommen werden. In beiden Klassen wurden Bezeichner (Attribute, Parameter) angepaßt und einige nicht benutzte Methoden entfernt. Die `find`-Methode aus `Tessi.ExtTextPane` wurde neu implementiert. Die direkte Erzeugung von Instanzen der `Document`-Implementierung wurde in eine *Factory* verlegt. Eine einfache Verbesserung des Moduls, die mit einer anderen `Document`-Klasse auskommt, wird so leicht möglich. Diese Vorkehrungen nützen natürlich nichts, wenn die Schnittstelle zum `DataRespository` geändert wird (siehe Abschnitt 5.1.6 zur Kritik der Schnittstelle zum Spezifikationstext).

Die benötigten `Actions` wurden wieder in eigenen Klassen implementiert. Die Namen der Klassen haben das Suffix `Action`.

Dem Editor wurde ein Kontextmenü hinzugefügt. Es enthält Befehle, welche Ereignisse zum Erzeugen von Modellelementen auslösen. Diese Befehle sollen entsprechende Dialoge öffnen, welche vom Modul „Modellmanipulation“ bereitgestellt werden. Die Menübefehle sind deaktiviert, wenn kein Text markiert wurde.

[7] Die Datei `01-12-02.xml` ist ein offizielles OMG-Dokument. Sie enthält die XMI-Beschreibung des *UML Interchange Model* (siehe dazu [Tos02, Abschnitt 3.2.1]). Die Datei `01-12-02_Diff.xml` enthält JMI-Annotationen für die UML 1.3. Diese Datei stammt aus dem MDR-Projekt. Beide Dateien wurden von dort bezogen.

6.8 Modul Modellübersicht

Packages: `de.tuc.tessi.modelOverview`
Moduldatei: `modelOverview-1.0.jar`

Die Baumansicht ist ähnlich zu der aus TESSI 1.1. Es wurden jedoch einige Veränderungen vorgenommen. Die Ansicht zeigt unter einem Wurzelknoten mit dem Namen des Projektes Kategorieknoten für Klassen, Assoziationen, Generalisierungen, Zustandsautomaten und Kollaborationen. Darunter sind jeweils die den Namen entsprechenden Modellelemente angeordnet. Knoten für Modellelemente enthalten Kindknoten, die modellierten Eigenschaften entsprechen. Bei Klassen sind dies zum Beispiel Assoziationen, Vererbungsbeziehungen, Attribute, Methoden, Zustandsautomaten und Kollaborationen.

Die Implementierung der Baumansicht verläßt sich, wie in TESSI 1.1, stark auf die von Java Swing bereitgestellten Klassen. Für die Kategorien und alle Modellelemente wurden eigene Knotentypen implementiert, die von der Swing-Klasse `DefaultMutableTreeNode` abgeleitet sind und einige TESSI-spezifische Eigenschaften hinzufügen (z. B. eigene Icons). Für die Logik, die die Anordnung der Knoten im Baum kontrolliert (im Swing-Jargon *tree model* genannt), gilt dies entsprechend.

Für die verschiedenen Knotentypen sind Kontextmenüs definiert, welche die in Abschnitt 5.5 vorgegebenen Befehle enthalten.

6.9 Modul Modellmanipulation

Packages: `de.tuc.tessi.model`
Moduldatei: `model-1.0.jar`

Für die Eingabe von Eigenschaften wurden Komponenten implementiert, aus denen die Dialoge aufgebaut wurden. Die Eigenschaftsdialoge werden sowohl für die Bearbeitung bestehender, als auch zum Anlegen neuer Modellelemente verwendet. Der Nutzer kann nahtlos zwischen beiden Funktionen wechseln. Die Form der Dialoge wurde Dialogen in gängigen CASE-Werkzeugen angenähert.

Die Expansionsdialoge aus TESSI 1.1 wurden weitgehend übernommen. Allerdings kann das Modell nicht mehr direkt über diese Dialoge beeinflußt werden. Statt dessen werden die Eigenschaftsdialoge geöffnet. Der auslösende Expansionsdialog bleibt dabei offen, so daß die „Zurück“-Navigation entfallen konnte.

Während die Bearbeitung der Elementeeigenschaften direkt in den Dialogen implementiert ist, wird das Löschen von Modellelementen an nur einer Stelle implementiert (`ModelEventAction`).

6.10 Modul Thesaurus

Packages: `de.tuc.tessi.thesaurus`
Moduldatei: `thesaurus-1.0.jar`

Für dieses Modul wurden die Packages `de.tuc.isse.tessi.wordnet` und `de.tuc.isse.tessi.thesaurus` unverändert übernommen. Sie wurden als ein JAR-File (`thesauruslib.jar`) in die Moduldatei integriert.

Die Eingabemaske für Suchoptionen wird von der Klasse `ThesaurusPanel` implementiert. Die Steuerelemente und die erzeugten Suchoptionen orientieren sich am Thesaurusmenü von TESSI 1.1. Der eingesetzte Code ist teilweise auch von dort übernommen und angepaßt worden.

Das Textfenster zum Anzeigen der Suchergebnisse ist ein nur wenig erweiterter `javax.swing.JTextPane`. Hinzugefügt wurde nur eine Funktion zur Hervorhebung des Suchbegriffes. Grundzüge davon stammen ebenfalls von TESSI 1.1.

6.11 Modul Textgenerierung

Packages:	`de.tuc.tessi.textGeneration`
Moduldatei:	`textGeneration-1.0.jar`

Die drei Textformate werden jeweils von einer Klasse generiert, deren Namen sich aus der Bezeichnung des Textformates und dem Suffix `TextGenerator` ergeben (Klassen `StaticModelTextGenerator`, `DynamicModelTextGenerator` und `MetricsTextGenerator`). Die ersten beiden Klassen entstanden zunächst aus der Zerlegung der Klasse `Tessi.DRText` von TESSI 1.1. Sie wurden dann an das UML-Datenmodell angepaßt. Bei der Umstellung wurde auch eine gemeinsame Superklasse (`AbstractTextGenerator`) extrahiert. Die dritte genannte Klasse `MetricsTextGenerator` entstand analog aus der TESSI 1.1-Klasse `Tessi.Metrics`.

Die Anzeige aller generierten Texte wird durch einen nur gering erweiterten `javax.swing.JTextPane` realisiert. Hinzugefügt wurde eine Anpassung an die Generatoren als Quelle des angezeigten Inhaltes.

6.12 Modul Hilfe

Packages:	`de.tuc.tessi.help`
Moduldatei:	`help-1.0.jar`

Die Klasse `ShowHelpAction` implementiert die Reaktion auf das Standardereignis `ShowHelpRequestEvent`. Sie führt eine Umwandlung der erhaltenen Hilfe-ID durch und ermittelt daraus eine URL, die sie einem externen Internetbrowser zur Ansicht übergibt.

Für Hilfeidentifikatoren wurden einige Festlegungen getroffen:

- Hilfeidentifikatoren sind Zeichenketten (`java.lang.String`)
- Eine Zeichenkette, die mit den Zeichen „`HID:`“ beginnt, bezeichnet eine wohldefinierte Position in den bekannten Hilfetexten. Die Definition dieser Positionen geschieht durch die Datei `help-ids.properties`. Wie alle Konfigurationsdateien wird sie im Verzeichnis `$tessi.base.dir/config` erwartet. Festgelegt sind bisher folgende IDs:
  - `HID:TOP` bezeichnet die Einstiegsseite zur Hilfe von TESSI 2.0
  - `HID:TOC` bezeichnet das Inhaltsverzeichnis der Hilfe zu TESSI 2.0
  - `HID:ABOUT` bezeichnet eine Seite mit Informationen zum Programm TESSI 2.0

  - `HID:APIDOC` bezeichnet die Einstiegsseite zur API-Dokumentation

- Eine Zeichenkette, die mit den Zeichen „http://“ beginnt wird als URL interpretiert.
- Jede andere Zeichenkette wird als relativer Pfad unter dem Hilfeverzeichnis `$tessi.base.dir/doc/help/` interpretiert.

Module tragen zur Hilfe bei, indem sie ihre Hilfetexte in einem Unterverzeichnis im Verzeichnis `$tessi.base.dir/doc/help` ablegen. Der Name des Unterverzeichnisses sollte dem Namen der Moduldatei ohne Versionsinformation entsprechen.

Zusätzlich zu den Texten ist eine Datei mit dem Namen `toc.html` in dem Unterverzeichnis abzulegen. Diese Datei enthält das Inhaltsverzeichnis der Hilfetexte des Moduls. Es handelt sich dabei nicht um eine gültige HTML-Datei. Sie enthält auf jeder Zeile einen Link zu einem Thema der Hilfe. Die Zeilen haben die Form:

```
<a class='class_def' href='url'>url_text</a> <br/>
```

mit:

`class_def`	Ein Klassenidentifikator für den Link. Mögliche Werte sind `toc_link_l1`, `toc_link_l2` oder `toc_link_l3`. Mit Hilfe eines Stylesheets werden über diese Identifikatoren Formatierungen vorgenommen.
`url`	Eine relativer Pfad oder eine URL zum Ziel des Links.
`url_text`	Eine Erläuterung des Links.

Diese Dateien werden vom Hilfemodul ausgewertet, um HTML-Seiten für die Hilfeidentifikatoren HID:TOP und HID:TOC zu generieren. Fehlt die Datei `toc.html` in einem Unterverzeichnis von `$tessi.base.dir/doc/help`, dann werden die Hilfetexte in diesem Verzeichnis nicht in die Hilfe aufgenommen. Sie sind jedoch trotzdem über ihren relativen Pfad aufrufbar.

6.13 Adapterbibliothek

Die Bibliothek besteht aus den beiden Packages `de.tuc.tessi.umlAdapter` und `de.tuc.tessi.umlAdapter_1_3`. Sie wurden in das JAR-File `umlAdapter.jar` gepackt, das sich in den `lib`-Unterverzeichnissen von *Build*- und Programmumgebung befindet.

Die Adapterklassen sind zustandslos, d. h. sie speichern keine Daten zwischen und lauschen auch nicht auf Ereignisse aus dem Repository. Es sollte keine zusätzliche Schicht über dem Repository geschaffen werden, wie das z. B. bei den in [Tos02, Abschnitt 4.2] beschriebenen Adapterklassen der Fall war. Es wird auch nicht sichergestellt (bzw. es wird gar nicht geprüft), daß zu einem Objekt im Repository genau eine Instanz eines Adapters gehört. Das ist auch nicht nötig, da die Methoden immer direkt in das Repository durchgreifen und für den Klienten im Verhalten kein Unterschied feststellbar ist. Methoden, die für Vergleiche und eine rudimentäre Visualisierung eingesetzt werden, wurden überschrieben. Die Methoden `toString`, `compareTo` und `equals` arbeiten nur

mit Eigenschaften der Repositoryobjekte. Dadurch zeigen zwei verschiedene Instanzen eines Adapters, die auf dasselbe Repositoryobjekt verweisen, das gleiche Verhalten wie nur eine einzige Instanz.

Ein Nebeneffekt der Adapterbibliothek ist, daß es mit ihrer Hilfe möglich ist, kleinere Unterschiede zwischen den Versionen der UML zu verbergen. Dadurch könnte man Modelle verschiedener UML-Versionen (nacheinander) in TESSI bearbeiten, solange deren Unterschiede die Möglichkeiten der Bibliothek nicht sprengen.[8] In TESSI 2.0 ist diese Möglichkeit vorbereitet. Dazu wurden Schnittstellen und Implementierung der Adapterbibliothek voneinander getrennt und eine Implementierung für die UML 1.3 programmiert. Implementierungen für weitere UML-Versionen können in späteren Versionen hinzugefügt werden.

Welche Implementierung gerade aktiv ist, bleibt den übrigen Modulen verborgen. Sie müssen sich eine Referenz auf die Adapterbibliothek über einen Aufruf an eine übergeordnete *Factory* besorgen. Dort ist momentan die Implementierung für die UML 1.3 fest eingestellt. Zukünftig, wenn andere Implementierungen benötigt werden und vorliegen, kann das Modul „Datenhaltung“ die benötigte Implementierung nach dem Laden aktivieren. Dazu sind nur geringe Änderungen nötig.

Problematisch ist hieran nur die unmögliche Koexistenz verschiedener Klassen mit gleichem Namen. Experimentell wurden daher TESSI-spezifische Annotationen in ein XMI-Differenzfile eingetragen. Es gelang ohne weiteren Aufwand, Interfaces für die UML 1.3 und 1.4 zu generieren, die jeweils in verschiedenen Packages untergebracht sind. So kann man dem Namensproblem aus dem Weg gehen. Die Differenzdateien wurden im Konfigurationsverzeichnis untergebracht. Beim Aufruf des *Buildtargets* `interfaces` werden standardmäßig auch die Dateien `uml-1.3.jar` und `uml-1.4.jar` mit erzeugt, die die veränderten Interfaces enthalten.[9]

Nachteilig dagegen ist, daß eine weitere Abhängigkeit zwischen den Modulen hinzukommt: nämlich die von der Adapterbibliothek. Allerdings wurde die Schnittstelle zur Datenhaltung (Interface `DataRepository`) nicht verändert. Module sind außerdem nicht gezwungen, die Bibliothek zu benutzen.[10] Man kann sie als zwischenschaltbaren Filter sehen, der einen Teil der Komplexität der UML-Struktur verbirgt.

[8] Für MDR stellt diese Aufgabe ohnehin kein Problem dar.
[9] Zum *Build*-Vorgang siehe Abschnitt 6.1.1 auf Seite 76.
[10] Die Adapterbibliothek selbst nutzt nur die Schnittstelle `DataRepository`.

7. EINSCHÄTZUNG UND AUSBLICK

7.1 Analyse von TESSI 2.0

Der Analyse von TESSI 1.1 aus Abschnitt 5.1 muß eine ähnliche Analyse von TESSI 2.0 folgen, um die Veränderung zum Guten oder Schlechten festzustellen und zu dokumentieren. Diese Analyse wurde durchgeführt, ihre Ergebnisse werden im Folgenden vorgestellt.

7.1.1 Modularisierung

Bei der Untersuchung von TESSI 1.1 wurde zunächst versucht, Module zu entdecken. Diese Aufgabe kann hier entfallen, da TESSI 2.0 ausdrücklich modular entworfen wurde. Es ist höchstens zu prüfen, ob die Module wirklich gekapselt sind, so daß sich die Beziehungen ausschließlich auf die beschriebenen Beziehungen beschränken.

Diese Prüfung kann mit z. B. JDepend[1] durchgeführt werden. Das Werkzeug erstellt einen Report verschiedener Eigenschaften der Packages. Der Report enthält auch Angaben darüber, von welchen anderen Packages die Klassen eines Packages abhängen. Eine alternative Möglichkeit wäre der Einsatz des Werkzeugs Macker.[2] Im Gegensatz zur Erzeugung vieler Informationen in einem Report, der erst interpretiert werden muß, kann man von Macker einfach die Aussage „Alles o.k.“ (oder eben nicht) erhalten.[3]

Alle Module in TESSI 2.0 sollten nur vom Basismodul abhängig sein, da dieses die grundlegenden Dienste bereitstellt. Eine weitere Abhängigkeit darf zum Package `de.tuc.tessi.umlAdapter` bestehen, da es sich hierbei um eine unterstützende Bibliothek und nicht um ein Modul handelt.[4]

Der Report der Analyse mit JDepend bestätigt, daß außer den dokumentierten Abhängigkeiten, keine weiteren Beziehungen zwischen den Modulen bestehen.[5] Das angestrebte Ziel, durch die Einführung von Modulen die Beziehungen zwischen den Bestandteilen von TESSI zu vereinfachen und überschaubarer zu machen, wurde also erreicht.

---

[1] Siehe Beschreibung in Abschnitt 2.3.2 auf Seite 27.

[2] Siehe Kurzbeschreibung in Abschnitt 2.3.4 auf Seite 28.

[3] Die Informationen des Reports von JDepend wurden allerdings auch an anderer Stelle gebraucht. Daher wurde JDepend der Vorzug vor Macker gegeben.

[4] Analog dazu bestehen selbstverständlich auch Abhängigkeiten zu Packages anderer Bibliotheken, z. B. zu den Packages der Klassenbibliothek von Java und zu Netbeans MDR.

[5] Der Report von JDepend wurde wegen seiner Länge von rund 16 Seiten nicht mit in die Arbeit aufgenommen. Er kann jedoch jederzeit über das *Buildtarget* `metrics` neu erzeugt werden. Graphisch aufbereitet, und daher im Überblick aussagekräftiger, ist jedoch die Übersicht, die das JDepend-Plugin zu Eclipse erzeugt.

7.1.2 Separation of Concerns

In der Analyse von TESSI 1.1 wurde die Qualität der Separierung verschiedener Aspekte untersucht. Die Untersuchung soll nun unter den gleichen Bedingungen mit TESSI 2.0 durchgeführt werden.

Aspekt Modelldatenmanipulation

Erkennung Die Untersuchung soll der Übersichtlichkeit halber wieder nur anhand des Modellelementes „Klasse“ verdeutlicht werden. Eine Instanz einer Klasse wird über die JMI-Schnittstellen durch einen Aufruf an den Klassenproxy des Modellelements `Class` erzeugt. Zerstört wird die Instanz durch den Aufruf ihrer `refDelete`-Methode. Damit sind prinzipbedingt bereits zwei Instanzen in die Realisierung des Lebenszyklus einbezogen. Während die Erzeugung durch die Methode `createClass` leicht erkannt werden kann, ist dies bei der Zerstörung schwieriger. Über eine Methode `refDelete` verfügen nämlich alle Modellelemente. Hier muß die Suche typbasiert durchgeführt werden.

Die Eigenschaften der Modellelemente werden jeweils durch `set`-Methoden verändert. Die `set`-Methoden, die für Klassen interessant sind, werden in insgesamt sechs Interfaces deklariert (das Interface für `Class` selbst sowie Interfaces, die in der Vererbungshierarchie über `Class` stehen). Auch hier muß der Typ in die Suche einbezogen werden.

Alle Module von TESSI 2.0 verwenden die Adapterbibliothek. Gesucht werden muß daher nicht nach den JMI-Schnittstellen, sondern nach den Adapterschnittstellen. Die Struktur der Interfaces der Adapterbibliothek und deren Methoden sind jedoch an die JMI-Schnittstellen angelehnt, so daß die eben gemachten Angaben nur leicht angepaßt werden müssen.

Analyse und Ergebnis Abbildung 7.1 zeigt das Ergebnis der Suche mit AMT. Die typbasierten Fundstellen werden grau angezeigt, da sie ohne ausdrucksbasierte Fundstellen wertlos sind. Aus der Resultatmenge wurden deshalb auch alle Klassen entfernt, die lediglich typbasierte Fundstellen aufwiesen.

Die beiden Säulen gehören zu den Klassen `ClassDialog` (linke Säule) und `ModelEventAction` aus dem Package `de.tuc.tessi.model`. `ClassDialog` implementiert einen Eingabedialog für die Erzeugung und Bearbeitung von Klassen im Modell. `ModelEventAction` ist im Modul „Modellmanipulation“ verantwortlich für die Behandlung aller `ModelEvent`s.

Die Erzeugung von Instanzen und das Verändern von Eigenschaften der Instanzen (beides erkennbar an den zweifarbigen Linien) konzentrieren sich auf die Klasse `ClassDialog`. In der Klasse `ModelEventAction` dagegen wird die Zerstörung von Instanzen aller Modellelemente als Reaktion auf die entsprechende Nutzereingabe durchgeführt. Der Aufruf der `delete`-Methode erfolgt an einer Referenz vom Typ `ModelElementAdapter`, hinter der sich (polymorph) alle Modellelemente verbergen können. Durch die verwendeten Suchparameter (siehe Eingabezeilen unten im Bild 7.1) ist diese Zeile nur einfarbig.

Der gesuchte Code ist auf zwei Klassen verteilt, die sich im selben Modul befinden. Von einer vollständigen Separierung *in eine Klasse* kann man daher nicht sprechen. Gegenüber TESSI 1.1 wurde jedoch der Vorteil erzielt, daß die Reaktion auf Nutzereingaben nicht an verschiedenen Stellen dupliziert vorkommt. Außerdem konzentriert sich die Modellmanipulation in TESSI 2.0 auf

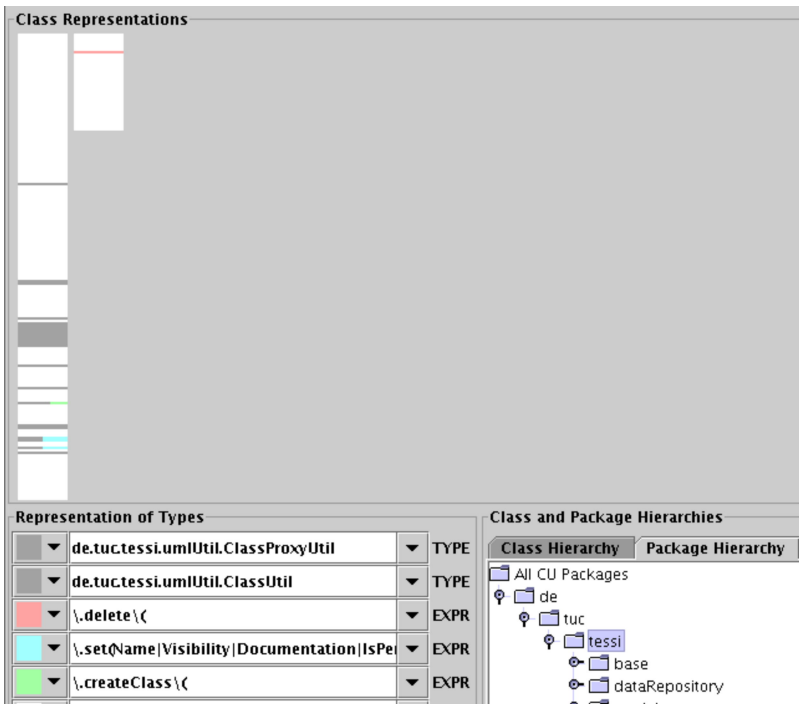

Abb. 7.1: Untersuchung zur Verteilung von Codefragmenten zur Manipulation von Klassen in TESSI 2.0

ein Modul.

Eine bessere Separierung läßt sich mit herkömmlichen Mitteln schwer erreichen. Man könnte die Zerstörung von Instanzen von Klassen ebenfalls dem Dialog übertragen. Diese Aufgabe ist jedoch eigentlich nicht die Verantwortlichkeit dieser Klasse. Das Verändern der Eigenschaften könnte in die `Action` verschoben werden. Dazu müßte jedoch ein Hilfskonstrukt eingeführt werden, um die Nutzereingaben zwischenzuspeichern. Beide Lösungen sind nach Ansicht des Autors schlechter als die gewählte Lösung.

Aspekt Modelldatenvisualisierung

Erkennung Wie bei der Untersuchung von TESSI 1.1 wird auch hier versucht, die visualisierenden Codefragmente an der Abfrage von Eigenschaften von Modellelementen zu erkennen. Gesucht wird nach `get`-Methoden für die Eigenschaften. Die Suche wird auch typbasiert durchgeführt.

Nicht betrachtet werden Klassen aus dem Modul „Textgenerierung“. Sie greifen zwar ebenfalls auf die Modellelemente zu und stellen sie in gewisser Weise auch „graphisch“ dar (durch Texte). Nach der Beschreibung auf Seite 45 gehören sie jedoch nicht zu diesem Aspekt.

Analyse und Ergebnis Das Ergebnis der Suche zeigt Abbildung 7.2. Obwohl andere Farben verwendet wurden und obwohl anderer Quellcode untersucht wurde, ist das Ergebnis visuell ähnlich (vgl. Abbildung 4.4 auf Seite 46).

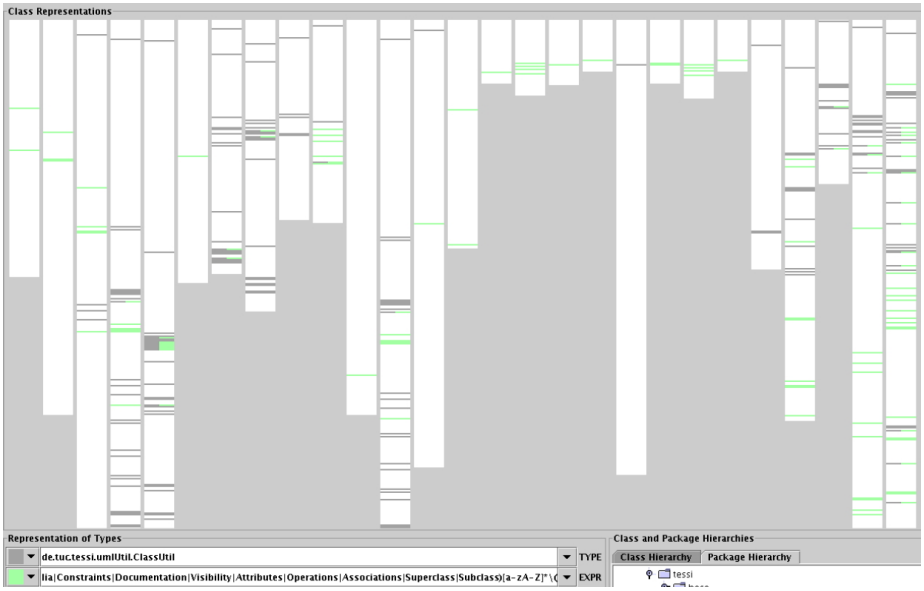

Abb. 7.2: Verteilung von Codefragmenten zur Abfrage von Eigenschaften von Klassen in TESSI 2.0

Die Säulen im Bild gehören zu Klassen aus den Modulen „Modellübersicht“, „Modellmanipulation“ und „Spezifikationstext“. Es ist deutlich, daß der Code nicht nur auf mehrere Klassen verteilt ist, sondern sich auch über mehrere Module erstreckt. TESSI 2.0 bringt also keine Verbesserung der Separierung dieses *concerns*.

Aspekt Textgenerierung

Erkennung Für Textgeneratoren definiert das Modul „Textgenerierung“ ein Interface (`TextGenerator`). Dieses deklariert nur eine Methode mit Namen `generate`, welche das Generieren veranlaßt. Das Interface wird von einer abstrakten Klasse (`AbstractTextGenerator`) implementiert, die drei konkrete Kindklassen besitzt (für jede Textart eine). Der Aufruf der Methode kann verwendet werden, um die Steuerung der Funktion aufzufinden. Die Implementierungen der Methode zeigen die „Einstiegspunkte“ zu den Generatoren an. Die Methoden, die tatsächlich Text generieren, sind ausnahmslos `protected`. Eine Suche nach dem Aufruf solcher Methoden liefert keine zusätzlichen Informationen.

Analyse und Ergebnis Abbildung 7.3 zeigt das Resultat der Suche. Gesucht wurde wieder kombiniert typbasiert und ausdrucksbasiert, obwohl dieses Vorgehen hier eigentlich nicht nötig war. Die fünf Säulen im Bild gehören zum Interface `TextGenerator` (ganz rechts), zur Klasse `AbstractTextGenerator` (ganz links), zu den beiden Textgeneratoren `StaticModelTextGenerator` (2. von links) und `DynamicModelTextGenerator` (4. von links) sowie zur Klasse `GeneratedTextViewer`, die eine graphische Komponente zum Anzeigen der Tex-

te implementiert (Säule in der Mitte).

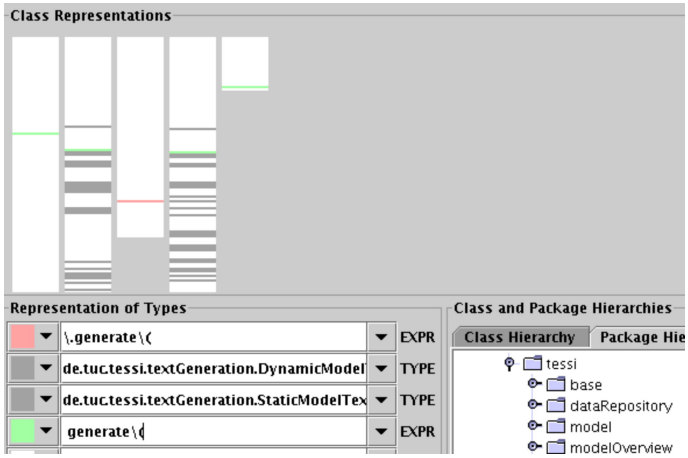

Abb. 7.3: Untersuchung zur Verteilung von Codefragmenten zur Textgenerierung in TESSI 2.0

Der Code zur Generierung einer Textart ist jeweils in nur einer Klasse bzw. der Superklasse `AbstractTextGenerator` untergebracht. Die Behandlung des Aspekts kann man daher als separiert betrachten.

Im Vergleich zu TESSI 1.1 zeigt sich der Unterschied, daß für jede Textart nun eine eigene Klasse vorhanden ist. Diese Klassen sind hauptsächlich aus der Zerlegung der TESSI 1.1-Klasse `Tessi.DRText` entstanden. Die Lösung in TESSI 2.0 spiegelt die Verantwortlichkeiten der Klassen wider, im Gegensatz zur alten Lösung, wo die Generatorklasse noch zusätzliche Aufgaben hatte. Nach Ansicht des Autors wird der Aspekt in TESSI 2.0 weiterhin separat behandelt, nur die Struktur ist einfacher und verständlicher geworden.

7.2 Zukünftige Verbesserungen

Bei der Erstellung dieser Arbeit wurden einige Möglichkeiten für Verbesserungen festgestellt, die jedoch außerhalb des Rahmens dieser Arbeit lagen. Sofern möglich und nötig, wurden diese Veränderungen jedoch vorbereitet. In diesem Abschnitt sollen sie vorgestellt werden.

7.2.1 Unterstützung verschiedener UML-Versionen

In den Abschnitten 5.10 und 6.13 wurde die Adapterbibliothek und deren Implementierung beschrieben. Es wurde angedeutet, daß mit Hilfe der Adapter die Unterstützung für Modelle zu verschiedenen Versionen des Metamodells UML eingebaut werden könnte. Voraussetzung dafür ist allerdings, daß die Unterschiede in den UML-Versionen durch die Adapter abgeschirmt werden können.

Um diese Unterstützung zu realisieren, muß für jede zu unterstützende UML-Version eine Implementierung der Adapterschnittstellen geschaffen werden. Diese befinden sich im Package `de.tuc.tessi.umlAdapter`. Es ist anzunehmen, daß viele Veränderungen zwischen UML-Versionen nur einen Teil der in TESSI

benutzten Modellelemente betreffen. Der Aufwand dürfte sich also in Grenzen halten.

Die Adapterbibliothek beinhaltet eine *Factory*, mit deren Hilfe Benutzer den „Einstiegspunkt“ zur Bibliothek, also eine Referenz auf die Adapterfabrik der aktiven Implementierung erhalten. In diese Klasse muß das Aktivieren unterschiedlicher Implementierungen noch eingefügt werden.

Alle Implementierungen sollten in das zur Bibliothek gehörende JAR-File `umlAdapter.jar` integriert werden.

Darüber hinaus bedarf es noch kleinerer Änderungen am Code des Moduls „Datenhaltung“. Beim Laden einer Projektdatei ist die verwendete UML-Version festzustellen und der `UmlAdapterFactoryFactory` mitzuteilen.

Für das *Metadata Repository* (MDR) stellt die Unterstützung verschiedenster Metamodelle kein Problem dar. Voraussetzung ist lediglich, daß für das Metamodell eine XMI-Beschreibung vorliegt. Das bedeutet implizit, daß das Metamodell eine Instanz des MOF Metametamodells sein muß.

7.2.2 Unterstützung verschiedener Thesauri

Die momentan vorhandene Unterstützung von Thesauri ist auf den Thesaurus WordNet zugeschnitten. Der Code für die Kapselung des Thesaurus (Klasse `de.tuc.isse.tessi.wordnet.WordNet` und zur Erzeugung der Anfragen stammt größtenteils von TESSI 1.1 (Autor ist Lars Rosenhainer).

Aus der Schnittstelle von `de.tuc.isse.tessi.wordnet.WordNet` sollte ein Interface extrahiert werden, welches auch für alternative Thesauri mächtig genug ist.[6] Implementierungen wären für die Ansteuerung des jeweiligen Thesaurus und die Anpassung der Schnittstellen verantwortlich. Die Erzeugung der Instanzen könnte einer *Factory* übertragen werden. Die Auswahl des zu verwendeten Thesaurus sollte konfigurierbar sein. Hilfreich für den Nutzer von TESSI wäre eventuell auch die Möglichkeit, mehrere Thesauri zur gleichen Zeit abfragen zu können.

7.2.3 Strukturierter Spezifikationstext

Es wurde bereits angedeutet, daß die momentan unstrukturierte Repräsentation des Spezifikationstextes künftigen Anforderungen eventuell nicht mehr genügt. Es soll hier keine Vermutung angestellt werden, wie der Spezifikationstext strukturiert repräsentiert werden könnte. Allerdings kann hier beschrieben werden, welche Bestandteile von TESSI 2.0 wie zu verändern wären.

Eine alternative Textrepräsentation könnte zwei Auswirkungen haben:

1. Die Behandlung des Textes wird nur im einzigen Klienten (Modul „Spezifikationstext“) verändert. Die Datenhaltung behandelt Text jedoch weiterhin nur als Zeichenkette.

2. Die Behandlung des Texts in TESSI wird durchgehend, von der Datenhaltung bis hin zu potentiell mehreren Klienten, verändert.

Im ersten Fall muß nur der Klient verändert werden. Der Teil der Schnittstelle zur Datenhaltung, der für den Spezifikationstext verantwortlich ist, bleibt

[6] Im Modul „Thesaurus“ ist die Klasse `Thesaurus` für die Kopplung zum Code aus TESSI 1.1 verantwortlich. Sie wäre eventuell ein besserer Kandidat für diese Schnittstelle.

unverändert. Die Nachteile der Schnittstelle, wie in Abschnitt 5.1.6 beschrieben, bleiben bestehen. Auch die Datenhaltung wird nicht verändert. Im Modul „Spezifikationstext“ wäre mit großer Wahrscheinlichkeit die Implementierung des Interfaces `javax.swing.text.Document` (die Klasse `UndoableStyledDocument`) zu ändern. Die Erzeugung von Instanzen dieser Klasse wurde bereits einer *Factory* übertragen. Hier müßte also nur an einer Stelle geändert werden.

Im zweiten Fall wird die Schnittstelle zur Datenhaltung verändert. Für die Beziehung von Datenhaltung zu Klienten des Spezifikationstextes sollte das *Document-View*-Muster (*design pattern*) benutzt werden. Dieser Fall bringt den größten Aufwand mit sich, hat aber das Potential, die erwähnten Probleme der Schnittstelle zu lösen.

Im Modul „Datenhaltung“ wäre zunächst die Behandlung des Textes (intern als *Byte-Array* realisiert) auf das neue, strukturierte Format umzustellen. Denkbar wäre XML als Format und DOM[7] als Repräsentation.[8] Die Klienten müssen nicht nur die neue Repräsentation des Spezifikationstextes übernehmen, sondern auch die neuen Eigenschaften des Formates für den Anwender nutzbar machen.

7.2.4 Unterstützung von Metriken

Einer der vom Modul „Textgenerierung“ erzeugten Texte gibt Auskunft über die Anzahl von Modellelementen verschiedener Kategorien. Der Inhalt des Textes sind also umfangsorientierte Metriken. Die Zuordnung dieses Textes zum Modul „Textgenerierung“ war nicht eindeutig (siehe dazu Seite 41). Eine Alternative zur Lösung in TESSI 2.0 wäre ein eigenes Modul „Metriken“. Für TESSI 2.0 wurde dies nicht getan, da das entstehende Modul sehr klein gewesen wäre. Außerdem ist die Generierung des Metrikentextes der Generierung der anderen Texte sehr ähnlich.

Allerdings könnte sich die Situation ändern. Denkbare Erweiterungen wären z. B. eine Auswertung der Metriken mit Auswirkung auf die Modelldaten oder die Erzeugung anderer Metriken. In einem solchen Fall könnte sich die Frage nach einem separaten Modul „Metriken“ wieder stellen. Dazu müßte das neue Modul erzeugt werden, wie in Anhang C beschrieben.

7.3 Fazit

In Abschnitt 4.3 wurde beschrieben, wie in der Vorbereitung dieser Arbeit die Architektur von TESSI 1.1 untersucht wurde und welche Ergebnisse die Analyse erbrachte. Speziell untersucht wurden der Modulaufbau (siehe Abschnitt 4.3.2) und die Qualität der Separierung von *concerns* (siehe Abschnitt 4.3.3).

Basierend auf der Analyse wurde anschließend eine Architektur für TESSI 2.0 entworfen (siehe Kapitel 5). Erkenntnisse aus der Analyse führten dazu, daß die Funktionalität auf insgesamt acht Module aufgeteilt wurde (siehe Abschnitt 5.1.1). Jedes Modul kann unabhängig vom Rest von TESSI 2.0 entwickelt und ohne Veränderung anderer Module ausgetauscht werden. Die komplexen Beziehungen zwischen Klassen in TESSI 1.1 sind nun auf wenige grundlegende Dienste zwischen Modulen beschränkt (siehe Abschnitte 5.1.2 bis 5.1.6).

[7] Document Object Model

[8] Dies ist eine Möglichkeit von mehreren. Es soll hiermit keine Aussage über eine besondere Eignung von XML und DOM gemacht werden.

Die entworfene Architektur wurde insgesamt in Kapitel 5 beschrieben, die dafür angefertigte Implementierung TESSI 2.0 in Kapitel 6. Dazu kommen ein Benutzerhandbuch (in elektronischer Form, siehe Anhang B) und eine Anleitung für Entwickler, die an TESSI 2.0 arbeiten werden (siehe Anhang C). Die Quellendistribution von TESSI 2.0 enthält außerdem die mit Javadoc realisierte Quellcodedokumentation. Mit diesen Dokumentationen wurden alle diesbezüglichen Punkte der Aufgabenstellung (siehe Seite 2) erfüllt.

Es konnten einige Verbesserung erreicht werden:

- Für TESSI existiert jetzt eine modulbasierte Architektur.
- Die entworfene Architektur wurde als TESSI 2.0 implementiert.
- Für Architektur, Quellcode und Erweiterung sowie für die Benutzung von TESSI 2.0 liegen Dokumentationen vor.

Einige Erweiterungen und Veränderungen, die für absehbar gehalten werden, wurden vorbereitet. In Abschnitt 7.2 wurde dies dokumentiert.

Es wurde allerdings auch festgestellt, daß auch TESSI 2.0 einige wünschenswerte Eigenschaften nicht aufweist. Genannt wurden z. B. die nicht optimale Schnittstelle der Datenhaltung für den Spezifikationstext. In der Analyse im Abschnitt 7.1.2 wurde weiter festgestellt, daß die Separierung zumindest einiger untersuchter Aspekte nicht vollständig ist.

Dieses Ergebnis überrascht nicht, denn es ist bekannt, daß mit den verwendeten Mitteln der Objektorientierung eine vollständige Separierung aller *concerns* nicht erreichbar ist. Ein mögliche Lösung für dieses Problem ist der Einsatz aspektorientierter Programmierung, was jedoch außerhalb des Umfanges dieser Arbeit lag. Eine zukünfige Arbeit könnte sich speziell mit der Verbesserung der Architektur aus diesem Gesichtspunkt befassen.

ABKÜRZUNGSVERZEICHNIS

API Application Programming Interface. Entwicklerdokumentation eines Softwaresystems. Zuerst in der Bedeutung der Dokumentation von Schnittstellen für Erweiterungen eines Programmes verwendet. Oft wird auch die gesamte Entwicklerdokumentation damit bezeichnet.

CASE Computer Aided Software Engineering. Begriff für die durch spezielle Software unterstützte Softwareentwicklung.

CORBA Common Object Request Broker Architecture. Von der OMG standardisierte Spezifikation einer Middleware zur Kopplung der Komponenten verteilter objektorientierter Systeme.

DOM Document Object Model. Das *Document Object Model* (DOM) ist eine plattform-unabhängige und sprachneutrale Schnittstelle, die Programmen und Skripten dynamisch den Zugriff und die Veränderung des Inhalts, der Struktur und des Stils von Dokumenten bereitstellt. (Übernommen und übersetzt von der DOM-Homepage http://www.w3.org/DOM/)

GUI Graphical User Interface. Graphische Nutzerschnittstelle.

IDE Integrated Development Environment. Integrierte Entwicklungsumgebung. In der Regel die Integration einer großen Anzahl von Programmierwerkzeugen unter einer einheitlichen Oberfläche.

IDL Interface Definition Language. Mit IDL wird eine Beschreibungssprache für Schnittstellen bezeichnet. Eine solche Sprache ist Teil der CORBA-Spezifikation.

JMI Java Metadata Interface. JMI ist ein kommender Standard, der eine Abbildung von Metamodell auf Java-Interfaces sowie Semantik für die Implementierung dieser Schnittstellen definiert.

LCOM Lack of Cohesion. LCOM ist eine objektorientierte Metrik. Es gibt verschiedene Formeln zur Bestimmung dieser Metrik (siehe z. B. [FAM99, Abschnitt 20.3.4]).

LOC Lines of Code. LOC ist eine einfache umfangsorientierte Metrik: die Anzahl der Codezeilen. Da Codezeilen in verschiedenen Programmiersprachen und je nach Programmierstil sehr unterschiedlich ausfallen können, muß meist genauer definiert werden, was eine Codezeile ist.

MOF Metaobject Facility. MOF ist eine von der OMG initiierte Spezifikation eines Metametamodells zur Beschreibung von Metamodellen (wie z. B. der UML) sowie die Transformation der (Meta-)Modellkonstrukte in CORBA IDL.

NCSS Non-Commented Source Statement. Bezeichnung für syntaktische Konstrukte im Quelltext, die für eine umfangsorientierte Metrik gezählt werden. Was genau NCSS sind, ist eine Frage der Definition und oft abhängig vom zählenden Werkzeug.

OMG Object Management Group. Industriezusammenschluß, der sich um die Entwicklung und Weiterentwicklung von Standards rund um objektorientierte Technologie bemüht. Beispiele für von der OMG standardisierte Spezifikationen sind CORBA und die UML.

UML Unified Modelling Language. Grafische Beschreibungssprache für die Modellierung von objektorientierten Softwaresystemen. Die UML ist ein mit Mitteln des MOF Metametamodells beschriebenes Metamodell.

XMI XML Metadata Interchange. Von der OMG standardisierte Spezifikation eines Austauschformates für objektorientierte Modelle. XMI basiert auf MOF (Metametamodell) und XML (Austauschformat).

XML Extensible Markup Language. Vom World Wide Web Consortium standardisierte Spezifikation einer Metasprache zur Beschreibung von Datenobjekten (sogenannte XML-Dokumente).

LITERATURVERZEICHNIS

[Bal00] Balzert, Helmut: *Lehrbuch der Software-Technik, Bd. 1.*. - Heidelberg; Berlin: Spektrum, Akad. Verlag, 2. Aufl. 2000. 10, 15, 30, 31, 33, 34

[FAM99] Bär, Holger; Bauer, Markus; Ciupke, Oliver; Demeyer, Serge; Ducasse, Stéphane; Lanza, Michele; Marinescu, Radu; Nebbe, Robb; Nierstrasz, Oscar; Przybilski, Michael; Richner, Tamar; Rieger, Matthias; Riva, Claudio; Sassen, Anne-Marie; Schulz, Benedikt; Steyaert, Patrick; Tichelaar, Sander; Weisbrod, Joachim: *The FAMOOS Object-oriented Reengineering Handbook*. http://www.iam.unibe.ch/ famoos/handbook/ 13, 17, 18, 19, 40, 94

[BM98] Brown, William J.; Malveau, Raphael C.; McCormick III, Hays W. "Skip"; Mowbray, Thomas J.: *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. New York, NY: John Wiley & Sons, 1998. 20

[Bus96] Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael: *Pattern-oriented Software Architecture: A System of Patterns*. Chichester, England: John Wiley & Sons, 1996. 20, 32, 34

[DDN03] Demeyer, Serge; Ducasse, Stéphane; Nierstrasz, Oscar: *Object-oriented Reengineering Patterns*. San Francisco: Elsevier Science (USA), 2003. 11, 13, 14, 18, 19, 32

[FH79] Fjeldstad, R. K.; Hamlen, W. T.: *Application program maintenance study: Report to our respondents*. Proceedings GUIDE 48, Philadelphia, PA., 1979. 14

[Fow99] Fowler, Martin: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. 13, 19, 32

[GoF96] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Bonn: Addison-Wesley-Longman, 1996. 20, 32, 55, 75

[GJ97] Ghezzi, Carlo; Jazayeri, Mehdi: *Programming Language Concepts*. John Wiley & Sons, 3. Auflage 1997. 38, 39

[Gem00] Gemeinhardt, Lars: *Verbindung zwischen TESSI und Rational Rose mittels XML*. Studienarbeit Technische Universität Chemnitz, 2000. 35, 36

[HK01] Hannemann, Jan; Kiczales, Gregor: *Overcoming the Prevalent Decomposition in Legacy Code*. In: Workshop on Advanced Separation of Concerns (Proceedings). International Conference on Software Engineering. Toronto (Kanada), May 2001. 24, 43

[Kau94] Kaufmann, Achim H.: *Software-Reengineering: Analyse, Restrukturierung und Reverse-Engineering von Anwendungssystemen*. München; Wien: Oldenbourg, 1994. 13, 18

[KGa95] Klösch, René; Gall, Harald: *Objektorientiertes Reverse Engineering*. Berlin; Heidelberg: Springer-Verlag, 1995. 10, 11, 12

[Kro97] Kroha, Petr: *Softwaretechnologie*. München: Prentice Hall, 1997. 10, 11, 13, 15, 16, 17, 18, 19, 30, 31, 48

[Lei00] Leidenfrost, Sven: *Erweiterung des Werkzeugs TESSI um Modellierungsmöglichkeiten der Sprache UML und Unterstützung von Interview-Fragen*. Diplomarbeit Technische Universität Chemnitz, 2000. 35, 37, 38

[Mar95] Martin, Robert C.: *Object Oriented Design Quality Metrics: An Analysis of Dependencies*. ROAD, Vol. 2, No. 3, Sep-Oct, 1995. 17, 19, 27

[Mue97] Müller, Bernd: *Reengineering – Eine Einführung*. Stuttgart: Teubner, 1997. 10, 11, 12, 13, 14, 15, 16, 21, 22, 38

[Oes01] Oesterreich, Bernd: *Objektorientierte Softwareentwicklung mit der UML*. - München; Wien: Oldenbourg Wissenschaftsverlag, 5. Aufl. 2001. 75

[Rug92] Rugaber, Spencer: *Program Comprehension for Reverse Engineering*. In Proceedings of the 1992 AAAI Workshop on AI and Automated Program Comprehension. San Jose, California, 1992. 14

[SSL01] Simon, Frank; Steinbrückner, Frank; Lewerentz, Claus: *Metrics Based Refactoring*. In P. Sousa and J. Ebert, editors, Proc. 5th European Conference on Software Maintenance and Reengineering, pages 30–38. IEEE, Los Alamitos, 2001. 19

[Str96] Strauß, Mathias: *Entwicklung und Implementierung eines Werkzeugprototypen für die objektorientierte Analyse und den Entwurf mit Aspekten des Reverse-Engineering*. Diplomarbeit Technische Universität Chemnitz-Zwickau, 1996. 35

[Tos01] Toschev, Jöran: *Metadatenaustausch mit XML Metadata Interchange (XMI)*. Seminararbeit Technische Universität Chemnitz, 2001. 35, 36, 37

[Tos02] Toschev, Jöran: *Prototyp eines UML-basierten Repositorys für TESSI*. Studienarbeit Technische Universität Chemnitz, 2002. 35, 36, 39, 40, 81, 84

[WBM94] Witt, Bernard I.; Baker, F. Terry; Merrit, Everett W.: *Software Architecture and Design: Principles, Models and Methods*. New York: Van Nostrand Reinhold, 1994. 31

ANHANG

A. STANDARDEREIGNISSE

Im Folgenden werden die in TESSI 2.0 definierten Standardereignisse aufgelistet. Der Name der implementierenden Klasse entspricht dem Ereignisnamen. Für jedes Ereignis sind das Package der Klasse, eine Kurzbeschreibung der Bedeutung des Ereignisses und mögliche Quellen angegeben. Die API der Klassen ist der API-Dokumentation zu TESSI 2.0 zu entnehmen.

`ApplicationCloseRequestEvent`

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers zur Beendigung der Applikation
Quelle:	Die Nutzereingabe wird durch Steuerelemente der Swing-Bibliothek registriert. Sie wird in das TESSI-Ereignis übersetzt. Menübefehl im Menü *File*

`ApplicationEvent`

Package:	`de.tuc.tessi.base`
Bedeutung:	Abstrakte Ereignisklasse. Kennzeichnet eine Klasse von Ereignissen, die der Steuerung der Applikation als Reaktion auf Nutzereingaben dient.

`AssociationChangedEvent`

Package:	`de.tuc.tessi.base`
Bedeutung:	Eigenschaften eines Modellelementes vom Typ `Association` haben sich geändert.
Quelle:	Wird als Reaktion auf Veränderungen im `DataRepository` erzeugt.

`AssociationEndChangedEvent`

Package:	`de.tuc.tessi.base`
Bedeutung:	Eigenschaften eines Modellelementes vom Typ `AssociationEnd` haben sich geändert.
Quelle:	Wird als Reaktion auf Veränderungen im `DataRepository` erzeugt.

AttributeChangedEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Eigenschaften eines Modellelementes vom Typ `Attribute` haben sich geändert.
Quelle:	Wird als Reaktion auf Veränderungen im `DataRepository` erzeugt.

ClassChangedEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Eigenschaften eines Modellelementes vom Typ `Class` haben sich geändert.
Quelle:	Wird als Reaktion auf Veränderungen im `DataRepository` erzeugt.

CloseProjectRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers zum Schließen des aktuellen Projektes.
Quelle:	Menübefehl im Menü *File*
Hinweis:	In TESSI 2.0 kann nur ein Projekt zu einer Zeit offen sein. Das Schließen dieses Projektes wirft die Frage auf, welches Projekt danach geöffnet ist. Es wurde festgelegt, daß in diesem Fall ein leeres Projekt erzeugt wird. Damit ist jedoch das Ereignis `CloseProjectRequestEvent` ein Synonym für `NewProjectRequestEvent`.

CopySelectionRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung, den aktuell markierten Text in die Zwischenablage zu kopieren.
Quelle:	Menübefehl im Menü *Edit*

CreateAssociationRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung, den Eingabedialog zur Erzeugung eines neuen Modellelementes vom Typ `Association` zu öffnen.
Quelle:	Befehl in Kontextmenüs verschiedener Module

CreateAttributeRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung, den Eingabedialog zur Erzeugung eines neuen Modellelementes vom Typ `Attribute` zu öffnen.
Quelle:	Befehl in Kontextmenüs verschiedener Module

CreateClassRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung, den Eingabedialog zur Erzeugung eines neuen Modellelementes vom Typ `Class` zu öffnen.
Quelle:	Befehl in Kontextmenüs verschiedener Module

CreateCollaborationRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung, den Eingabedialog zur Erzeugung eines neuen Modellelementes vom Typ `Collaboration` zu öffnen.
Quelle:	Befehl in Kontextmenüs verschiedener Module

CreateModelElementRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Abstrakte Ereignisklasse. Kennzeichnet eine Klasse von Ereignissen, die zum Anzeigen von Eingabedialogen zur Erzeugung von Modellelementen führen.

CreateOperationRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung, den Eingabedialog zur Erzeugung eines neuen Modellelementes vom Typ `Operation` zu öffnen.
Quelle:	Befehl in Kontextmenüs verschiedener Module

CreateStateMachineRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung, den Eingabedialog zur Erzeugung eines neuen Modellelementes vom Typ `StateMachine` zu öffnen.
Quelle:	Befehl in Kontextmenüs verschiedener Module

CutSelectionRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, den aktuell markierten Text in die Zwischenablage zu kopieren und anschließend im Text zu löschen.
Quelle:	Menübefehl im Menü *Edit*

DataInputOutputEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Abstrakte Ereignisklasse. Kennzeichnet eine Klasse von Ereignissen, die das Laden und Speichern von Projektdaten steuern.

DeleteModelElementRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, das aktuell markierte Modellelement zu löschen.
Quelle:	Befehle in Kontextmenüs verschiedener Module

DeleteSelectionRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, den aktuell markierten Textteil zu löschen.
Quelle:	Menübefehl im Menü „Edit“

EditModelElementRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, den Dialog zur Bearbeitung des aktuell markierten Modellelements zu öffnen.
Quelle:	Menübefehl in Kontextmenüs verschiedener Module

Event

Package:	`de.tuc.tessi.base`
Bedeutung:	Abstrakte Ereignisklasse. Superklasse aller Ereignisse in TESSI 2.0

ExportGeneratedTextsRequestEvent

Package:	`de.tuc.tessi.textGeneration`
Bedeutung:	Anweisung des Nutzers, die generierten Texte in Textdateien zu speichern.
Quelle:	Modulmenü des Moduls „Textgenerierung“

ExportModelToXMIRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, das aktuelle Modell in ein XMI-File zu speichern.
Quelle:	Modulmenü des Moduls „Datenhaltung“

ExportSpecificationTextRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, den aktuellen Spezifikationstext in ein Textfile zu speichern.
Quelle:	Modulmenü des Moduls „Datenhaltung“

FindEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Abstrakte Ereignisklasse. Kennzeichnet Ereignisse, die mit dem Finden von Textteilen zu tun haben.

FindNextRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, ausgehend von der aktuellen Caretposition das nächste Vorkommen (in Leserichtung) des zuletzt verwendeten Suchbegriffs zu finden.
Quelle:	Menübefehl im Menü *Edit*

FindPreviousRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, ausgehend von der aktuellen Caretposition das nächste Vorkommen (entgegen der Leserichtung) des zuletzt verwendeten Suchbegriffs zu finden.
Quelle:	Menübefehl im Menü *Edit*

FindRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, den Suchen-Dialog zu öffnen.
Quelle:	Menübefehl im Menü *Edit*

HighlightRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, den zuletzt verwendeten Suchbegriff im Text hervorzuheben.
Quelle:	Menübefehl im Menü *Edit*

ImportSpecificationTextRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, den aktuellen Spezifikationstext zu verwerfen und einen neuen Spezifikationstext aus einer Textdatei zu laden.
Quelle:	Modulmenü des Moduls „Datenhaltung“

ImportXMIModelRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, die aktuellen Modelldaten zu verwerfen und statt dessen neue Modelldaten aus einer XMI-Datei zu laden.
Quelle:	Modulmenü des Moduls „Datenhaltung“

ModelElementChangedEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Abstrakte Ereignisklasse. Kennzeichnet Veränderungen der Eigenschaften von Modellelementen im `DataRepository`.

ModelElementSelectionByUserEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Handlung des Nutzers: ein existierendes Modellelement wurde markiert.
Quelle:	Graphische Komponenten verschiedener Module

ModelEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Abstrakte Ereignisklasse. Kennzeichnet Ereignisse, die Modelldaten betreffen.

NewProjectRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, die aktuellen Projektdaten zu verwerfen und ein neues, leeres Projekt anzulegen.
Quelle:	Menübefehl im Menü *File*

OpenProjectRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, die aktuellen Modelldaten zu verwerfen und ein bestehendes Projekt aus einer Projektdatei zu laden.
Quelle:	Menübefehl im Menü *File*

OperationChangedEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Eigenschaften eines Modellelementes vom Typ `Operation` haben sich geändert.
Quelle:	Wird als Reaktion auf Veränderungen im `DataRepository` erzeugt.

ParameterChangedEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Eigenschaften eines Modellelementes vom Typ `Parameter` haben sich geändert.
Quelle:	Wird als Reaktion auf Veränderungen im `DataRepository` erzeugt.

PasteRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, den aktuellen Inhalt der Zwischenablage an die aktuelle Caretposition zu kopieren.
Quelle:	Menübefehl im Menü *Edit*

RedoRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, das letzte Rückgängigmachen einer Handlung zur Textbearbeitung rückgängig zu machen.
Quelle:	Menübefehl im Menü *Edit*

RefreshRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, den Inhalt aller Ansichten neu aufzubauen.
Quelle:	Menübefehl im Menü *Edit*

RepositoryChangedEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Benachrichtigung, daß die bisherigen Modelldaten komplett ungültig geworden sind und neue Modelldaten im Repository vorhanden sind
Quelle:	`DataRepository`

SaveProjectAsRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, die aktuellen Projektdaten in eine Projektdatei zu speichern. Der Name und Pfad der Datei wird vom Nutzer angegeben.
Quelle:	Menübefehl im Menü *File*

SaveProjectRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, die aktuellen Projektdaten in eine Projektdatei zu speichern, deren Name und Pfad durch einen vorherigen Lade- oder Speichervorgang bekannt ist.
Quelle:	Menübefehl im Menü *File*

SelectAllRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, den aktuellen Spezifikationstext komplett zu markieren.
Quelle:	Menübefehl im Menü *Edit*

ShowHelpRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, einen Hilfetext anzuzeigen.
Quelle:	Steuerungselemente an graphischen Komponenten aller Module; Menü *Help*

SpecificationTextChangedEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Benachrichtigung, daß der bisherige Spezifikationstext komplett ungültig geworden ist und daß ein neuer Spezifikationstext verfügbar ist.
Quelle:	`DataRepository`

StateChangedEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Eigenschaften eines Modellelementes vom Typ `State` haben sich geändert.
Quelle:	Wird als Reaktion auf Veränderungen im `DataRepository` erzeugt.

TestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Abstrakte Ereignisklasse. Kennzeichnet Ereignisse, die zum Testen verwendet werden.

TextEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Abstrakte Ereignisklasse. Kennzeichnet Ereignisse, die mit dem Spezifikationstext in Zusammenhang stehen.

TextManipulationEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Abstrakte Ereignisklasse. Kennzeichnet Ereignisse, deren Bearbeitung eine Veränderung des Spezifikationstextes bewirkt

TextSelectionByUserEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Benachrichtigung darüber, daß der Nutzer einen Textteil markiert hat.
Quelle:	Editor im Modul „Datenhaltung“

TextSelectionEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Abstrakte Ereignisklasse. Kennzeichnet Ereignisse, die das Markieren von Textteilen anzeigen.

UndoRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, die letzte Handlung zur Textbearbeitung rückgängig zu machen.
Quelle:	Menübefehl im Menü *Edit*

UnhighlightRequestEvent

Package:	`de.tuc.tessi.base`
Bedeutung:	Anweisung des Nutzers, die aktuelle Hervorhebung eines Suchbegriffes im Text aufzuheben.
Quelle:	Menübefehl im Menü *Edit*

B. BENUTZERHANDBUCH

Das Benutzerhandbuch ist Teil der Quellendistribution. Es liegt in Form einer Online-Hilfe vor und kann mit einem Internet-Browser angezeigt werden. Dazu ist die Datei `$tessi.base.dir/doc/help/index.html` zu laden. Direkt aus TESSI kann die Hilfe mit dem Menübefehl Help→Contents aufgerufen werden.

C. ENTWICKLERHANDBUCH

C.1 Erstellung eines neuen Moduls

Vorbereitung

Für ein neues Modul ist ein eindeutiger Packagename zu wählen. Dem Namen ist ein Präfix voranzustellen, das das Projekt und dessen Zuordnung zu einer Organisation deutlich macht. Der Name (ohne Präfix) sollte auch für das JAR-File verwendet werden, das als Container für das Modul dient. Der Name sollte um eine Versionsinformation ergänzt werden. Natürlich ist darauf zu achten, daß es zu keinen Namenskonflikten kommt (weder bei Ressourcen, noch beim JAR-File). Gegebenenfalls ist dazu beim Verwalter des Projektes TESSI nachzufragen.

***Beispiel**: Die Standardmodule von TESSI 2.0 bestehen in der Regel aus einem Java-Package, dessen Name eine sinngemäße Übersetzung des deutschen Modulnamens ist. Diesem Namen ist das Präfix* `de.tuc.tessi` *vorangestellt.*

Planung des Moduls

Der Autor eines Moduls muß einige wichtige Eckpunkte vorher festlegen und dokumentieren. Das betrifft mindestens:

- *Auf welche Ereignisse von außen soll das Modul reagieren? Reichen die Standardereignisse von TESSI 2.0 dazu aus?*

  In vielen Fällen sollten die Standardereignisse zur Steuerung ausreichen. Es ist die Menge an solchen Ereignissen festzustellen, die relevant für das neue Modul sind. Die Auswahl ist zu dokumentieren.

- *Welche Ereignisse soll das Modul emittieren? Welche Informationen sind als Parameter der Ereignisse relevant? Welcher Kategorie sind die Ereignisse zuzuordnen? Sind diese Ereignisse potentiell für andere Module interessant?*

  Ereignisse, die die Steuerung des Moduls von außen ermöglichen, können für andere Module nutzbar gemacht werden. Sie sind in diesem Fall zu dokumentieren. Die Funktion des Moduls kann so von anderen Modulen gesteuert werden. Dies gilt natürlich auch umgekehrt für die Steuerung anderer Module. Der Autor eines Moduls muß außerdem entscheiden, ob und wo in graphischen Komponenten seines Moduls der Anwender intuitiv Steuerungselemente für die Funktion anderer Module erwarten würde. Diese Frage läßt sich schwer allgemein beantworten. Ein Beispiel: Wird in einer graphischen Komponente eine Anzahl Klassen dargestellt, könnte der Nutzer ein Kontextmenü erwarten, welches ihm Zugriff auf den Eigenschaftsdialog der Klassen ermöglicht.

Werden Ereignisse neu eingeführt, stellt sich die Frage nach der Einordnung in die Klassenhierarchie unter `Event`. Das bedeutet, daß die neuen Ereignisklassen von geeigneten Superklassen abzuleiten sind. Die Entscheidung und die neuen Klassen sind zu dokumentieren.

- *Wie soll die Reaktion des Moduls auf Ereignisse von außen bzw. selbst emittierte Ereignisse aussehen?*

  Einerseits betrifft dies die Aktionen, die die eigentliche Funktion des Moduls ausmachen. Andererseits sollte nicht vergessen werden, daß eventuell auch die Reaktion auf Standardereignisse abgeändert werden muß. Ein klassisches Beispiel ist das Ereignis `ApplicationCloseRequestEvent`. Wenn zu einem Zeitpunkt aus der Sicht des Moduls das Beenden der Applikation zu schädlichen Konsequenzen führen würde (ungespeicherte Daten), muß das Modul die Behandlung des Ereignisses blockieren.

- *Welche graphische Komponenten werden benötigt, um das Modul zu repräsentieren? Welche Steuerungsmöglichkeiten soll der Nutzer zusätzlich erhalten (aus dem Modulmenü)?*

  Es ist zu überlegen, welche Komponenten zu erstellen sind und an welchen Stellen in der Oberfläche sie angezeigt werden sollen. Graphische Komponenten sind von der Klasse `javax.swing.JComponent` abzuleiten. Für den Inhalt der Komponenten und deren Steuerung ist der Autor des Moduls selbst verantwortlich. Werden mehr als eine graphische Komponente eingesetzt, könnte eine Umschaltlogik nötig werden. Das bedeutet, es könnten Situationen existieren, in denen als Reaktion auf eine Handlung in einer Komponente eine andere Komponente in den Vordergrund geschaltet werden soll.

  Ein spezifisches Modulmenü sollte dann definiert werden, wenn Aktionen des Moduls die Möglichkeiten der gemeinsamen Oberfläche übersteigen[1] oder wenn bestimmte Aktionen deutlich in der Oberfläche verankert werden sollen. Die Entscheidung liegt hier aber ebenfalls beim Modulautor.

Das neue Modul muß eine Klasse enthalten, die als `ModuleInitializer` fungiert. Diese Klasse muß das Interface `de.tuc.tessi.base.ModuleInitializer` implementieren.

Das neue Modul kann ein `ModuleServiceDirectory` enthalten. Es muß das Interface `de.tuc.tessi.base.ModuleServiceDirectory` implementieren. Momentan wäre dies jedoch nur sinnvoll, wenn das Modul das Standardmodul „Datenhaltung“ ersetzen soll, da ein `ModuleServiceDirectory` bisher nur dem Export der Datenhaltung aus diesem Modul dient.

Programmierung

Die Initialisierung eines Moduls zur Laufzeit beginnt mit dem Aufruf der Methode `ModuleInitializer.initializeModule`. Als Parameter wird eine Referenz auf das `ServiceDirectory` übergeben. Die Implementierung dieser Methode

[1] Ein Beispiel dafür ist das Menü „Datenhaltung“. Dieses ermöglicht das Laden und Speichern von einzelnen Kategorien von Projektdaten. Hier werden also mehr Möglichkeiten geboten als das Laden und Speichern der Projektdaten insgesamt (Befehl Datei→Speichern bzw. Datei→Öffnen).

ist der Platz für die Registrierung modulspezifischer Ereignisse, Aktionen und graphischer Komponenten. Vor der Registrierung müssen die entsprechenden Instanzen natürlich erzeugt werden.

Für graphische Komponenten gilt die Einschränkung, daß das oberste gruppierende Element vom Typ `javax.swing.JComponent` sein muß. Was dieses Element gruppiert und wie die Präsentationslogik aussieht, ist im Rahmen der Möglichkeiten von Java Swing nicht weiter eingeschränkt. Für die Graphik-Programmierung mit Swing sei auf die Hilfe zu Swing in der Dokumentation zu den JDKs verwiesen. Die JDK-Dokumentation enthält auch ein umfangreiches Tutorial zu Swing.

An Stellen, wo der Modulautor dies für nötig oder nützlich hält, sollten Steuerelemente für die Kontexthilfe eingefügt werden. Dabei ist darauf zu achten, daß diese Steuerelemente die erwartete Form haben. Die Steuerelemente sind mit dem `Command` für das Ereignis `ShowHelpRequestEvent` und einer passenden Hilfe-ID zu belegen. Natürlich sind auch die Hilfetexte entsprechend zu planen. Weitere Festlegungen für Hilfeidentifikatoren und Hilfedateien sind im Abschnitt 6.12 nachzulesen.

Hinweise zur Programmierung der einzelnen Dienste ist der API-Dokumentation zu TESSI 2.0 zu entnehmen. Die Dokumentation der Architektur und der Implementierungsdetails zu den Standardmodulen befinden sich in den Kapiteln 5 und 6. Als Beispiel für die Programmierung empfiehlt sich der Code der Standardmodule.

Bau und Installation

Alle zum Modul hinzugehörigen Ressourcen müssen in ein JAR-File gepackt werden. Sind weitere Ressourcen (Bibliotheken) enthalten, ist der `Classpath`-Eintrag im Manifest des JAR-Files entsprechend zu setzen.

Das Manifestfile des JAR-Files muß über gültige Einträge verfügen, die die Namen von Klassen angeben, die die in diesem Package definierten Ressourcen implementieren. Die Einträge folgen den Festlegungen für Manifestdateien. Die Einträge tragen den Namen der Klasse aus diesem Package, die die Ressource definiert. Der zugeordnete Wert entspricht dem vollständigen Klassennamen der implementierenden Klasse. Als Schlüssel sind momentan nur `ModuleInitializer` und `ModuleServiceDirectory` definiert.

***Beispiel**: Der Eintrag im Manifest für die Klasse `com.foo.bar.Baz` als `ModuleInitializer` sieht so aus:*

```
...
ModuleInitializer: com.foo.bar.Baz
...
```

Ein derart vorbereitetes Modul kann dann in das Modulverzeichnis von TESSI kopiert werden. Beim nächsten Start von TESSI wird es geladen und kann, fehlerfreie Funktion vorausgesetzt, benutzt werden.

Es ist günstig, ein möglichst parametrisiertes *Build*-Skript für den Bau des Moduls vorzusehen. Neben dem Bau des Moduls kann dieses Skript auch die Eintragungen im Manifest und sogar die Installation des JAR-Files ins Moduleverzeichnis übernehmen. Als Beispiel kann das *Build*-Skript zu TESSI 2.0 dienen.

C.2 Überarbeitung bestehender Module

Die Beschreibung der Funktionalität eines Moduls dient als Basis, auf der Autoren anderer Module aufbauen. Die wichtigste Information ist hier, welche Ereignisse von einem Modul dem Nachrichtensystem übergeben werden. Nicht-Standardereignisse sind zusätzlich zu dokumentieren.

Veränderungen am Basismodul mit von außen sichtbaren Auswirkungen haben sehr wahrscheinlich Einfluß auf alle anderen Module. Solche Veränderungen könnten z. B. die Schnittstellen der vier grundlegenden Dienste (siehe 5.1) betreffen. Sie sind natürlich entsprechend zu dokumentieren. Der Autor solcher Veränderungen sollte aufgrund des großen Einflusses seiner Handlungen prüfen, ob sie tatsächlich nötig sind. Ist dies der Fall, kann eventuell eine neue Schnittstelle eingeführt, die alte aber beibehalten werden. So kann verhindert werden, daß alle alten Module mit einem Schlag nicht mehr nutzbar sind.

Ein überarbeitetes Standardmodul muß mindestens die in Kapitel 5 definierte Funktionalität des Moduls beinhalten.[2] Erweiterungen des Funktionsumfanges können einfach durchgeführt werden, sind jedoch zu dokumentieren.

[2] Dies gilt natürlich nur, wenn der Funktionsumfang der Module erhalten bleiben soll.

D. QUELLTEXT

Der Quelltext zu TESSI 2.0 befindet sich zusammen mit der gesamten Quellendistribution auf einer CD, die dieser Arbeit beiliegt.

E. PROGRAMMÜBERGABE

Über die Übergabe des Programms TESSI 2.0 an die Professur Informationssysteme und Softwaretechnik wird ein Protokoll angefertigt. Das Dokument hält die Vollständigkeit des übergebenen Quellcodes, der Dokumentation und aller übrigen benötigten Bestandteile des Programms fest. Bei der Übergabe wird die Funktionsfähigkeit und Vollständigkeit der Programmfunktionen geprüft und das Ergebnis im Protokoll festgehalten. Das Dokument wird an der Professur hinterlegt.